Markt&Technik

Commodore-Amiga Inc.

AMIGA DOS-HANDBUCH

Die Entwickler-Dokumentation zum Amiga-DOS-Betriebssystem, Version 1.2. Für Anwender und Programmierer.

Für Amiga 500, 1000 und 2000.

AmigaDOS-Handbuch

Inhaltsverzeichnis

Zweites Buch: Das AmigaDOS-Programmierer-Handbuch

Drittes Buch: Das technische AmigaDOS-Handbuch

Vorwort

Das *AmigaDOS-Handbuch* enthält drei – im amerikanischen Original ursprünglich getrennt veröffentlichte – Bücher:

Das *AmigaDOS-Anwender-Handbuch*

das *AmigaDOS-Programmierer-Handbuch* und

das *Technische AmigaDOS-Handbuch*

Im *Anwender-Handbuch* finden Sie Informationen, die für alle Benutzer – also auch Einsteiger – von elementarer Wichtigkeit sind.

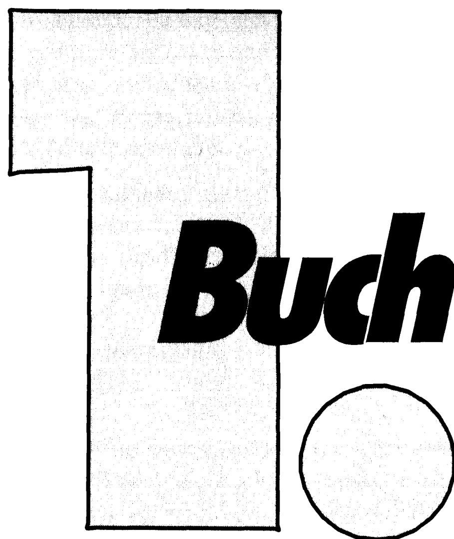
Der Amiga versteht nämlich weit mehr Instruktionen, als über die Workbench per Maus erreichbar sind. Diese Instruktionen können Sie verwenden, wenn Sie das sogenannte *CLI* (*Command Line Interface*) aktivieren.

Das *Programmierer-Handbuch* dagegen beschreibt, wie das AmigaDOS in selbsterstellten Programmen verwendet werden kann. Daneben beschreibt dieser Teil sowohl den Amiga Macro Assembler als auch den dazugehörigen Linker (Diese Programme sind im normalen Lieferumfang des Amiga nicht enthalten!).

Das *Technische AmigaDOS-Handbuch* vermittelt Ihnen den Aufbau und die Anwendung der im Betriebssystem intern verwendeten Datenstrukturen des AmigaDOS. Eine Beschreibung des Disketten-Aufzeichnungsformates fehlt ebensowenig wie Erklärungen zum Format der sogenannten *Objekt-Files* des AmigaDOS. Programmentwickler oder interessierte Anwender werden in diesem Buch Antworten auf viele programmtechnische Fragen finden.

AMIGA

DOS-Handbuch



Das Anwender-Handbuch

Einführung

Dieses Handbuch beschreibt das AmigaDOS und dessen Befehlsschatz. AmigaDOS ist das Floppy-Disk-Betriebssystem Ihres Amiga. Die Benutzer-Schnittstelle CLI (*Command Line Interface*) liest Ihre Tastatureingaben (AmigaDOS-Anweisungen) ein, stellt sie im CLI-Fenster dar und übergibt sie dem AmigaDOS, um sie vom Rechner ausführen zu lassen. In dieser Hinsicht ist das CLI eher eine der herkömmlichen Computer-Schnittstellen: Es werden Anweisungen eingegeben, die Schnittstelle zeigt sie auf dem Bildschirm an und macht sie für den Computer verständlich, dann werden sie ausgeführt.

Die sogenannte Workbench-Schnittstelle dagegen ist für den normalen Anwender aufgrund der Eingabe von Anweisungen über die Maus und Bildschirm-Icons angenehm zu bedienen und in ihren Möglichkeiten völlig ausreichend. Aber leider wird das DOS des Amiga durch die Workbench-Schnittstelle etwas eingeschränkt. Aus diesem Grund wurde die Workbench-Diskette mit der CLI-Schnittstelle ausgestattet, die dieses Manko rückgängig macht.

Um dieses Handbuch nutzen zu können, müssen Sie zunächst das CLI aktivieren. Eine solche Aktivierung bringt ein neues Icon mit dem Namen CLI auf Ihren Workbench-Bildschirm. Wenn Sie dieses Icon per Doppelklick anwählen, öffnet sich ein neues CLI-Fenster und Sie sind in der Lage, eigene Befehle per Tastatur einzugeben.

Die Aktivierung des CLI

Starten Sie Ihren Amiga mit der Workbench-Diskette (falls Sie einen Amiga 1000 besitzen, müssen Sie diesen zuvor natürlich auch die Kickstart-Diskette einlesen lassen). Als erstes öffnen Sie dann das Disketten-Icon der Workbench-Diskette durch zweimaliges Anklicken. Jetzt öffnet sich ein Fenster, das den Inhalt dieser Diskette anzeigt. Starten Sie nun das Programm PREFERENCES durch einen Doppelklick mit der Maus auf das entsprechende

Icon. Im linken unteren Drittel des Preferences-Fensters steht das Wort CLI mit je einem Knopf zum Ein- oder Ausschalten (ON/OFF). Klicken Sie den Schalter ON für Einschalten an. Im unteren rechten Teil des Preferences-Fensters finden Sie die Felder zum Speichern (SAVE) beziehungsweise für den vorübergehenden Gebrauch (USE), die Sie zum Verlassen des Programms PREFERENCES verwenden.

Das Öffnen eines CLI-Fensters

Um das CLI benutzen zu können, muß nun ein CLI-Fenster geöffnet werden. Öffnen Sie dazu die SYSTEM-Schublade. Das CLI-Icon, das Zeichen »1>« in einem Würfel, wird daraufhin sichtbar und kann durch einen Doppelklick mit der Maus geöffnet werden.

Der Gebrauch des CLI

Wenn CLI das aktive Fenster ist, können Sie nun die gewünschten Anweisungen über die Tastatur eingeben. CLI-Fenster können wie andere Fenster geformt und bewegt werden. Verwenden Sie zum Schließen des CLI-Fensters einfach den Befehl ENDCLI.

Workbench und CLI – Verwandtschaft und Unterschiede

Unter dem Begriff *Directory* verstehen wir das Inhaltsverzeichnis aller Dateien auf einer Diskette (oder in einem Unterverzeichnis der Diskette). Ein Directory kann ein oder mehrere Unter-Directories beinhalten, die ein oder mehrere Dateien unter einem Namen zusammenfassen. Mit der Anweisung DIR erscheint diese Liste der Dateien und Unterdirectories der augenblicklich benutzten Diskette auf dem Bildschirm – in unserem Fall eine Liste der Dateien, die von der Schnittstelle Workbench verwendet werden. Die Dateien und Directories entsprechen weitgehend den auf der Workbench sichtbaren Icons.

Geben Sie also direkt in das soeben geöffnete CLI-Fenster den Befehl DIR ein. Wenn Sie das mit DIR erschienene Directory genauer anschauen, werden Sie feststellen, daß sich in diesem Directory mehr Dateien befinden, als Icons unter Workbench anklickbar sind. Was Sie nicht wissen konnten: Die Workbench zeigt nur dann eine Datei X, wenn eine mit ihr verknüpfte Datei X.INFO existiert. Die Workbench verwendet die .INFO-Datei, um das Icon auf dem Bildschirm darzustellen und mit einer entsprechenden Datei zu verknüpfen.

Auf der Workbench-Diskette befindet sich zum Beispiel das Programm DISKCOPY. Es besteht aus zwei Dateien. Die Datei DISKCOPY enthält das Kopierprogramm und DISKCOPY.INFO die Information darüber, wie die Form des Icons aussieht und so weiter. Für den Amiga existiert zum Beispiel das Malprogramm *GraphiGraft*. Nehmen wir an, mit diesem Programm wurde eine Bildschirmgrafik erstellt und unter dem Namen MOUNT.PIC

abgespeichert. Für die Datei MOUNT.PIC bildet das Programm nun ein Icon. Dessen Form und die Information, daß MOUNT.PIC mit *GraphiGraft* erstellt wurde, werden in der Datei MOUNT.PIC.INFO abgelegt. Wenn Sie deshalb die Bilddatei MOUNT.PIC angeklickt und geöffnet haben, lädt die Workbench zuerst das Malprogramm *GraphiGraft* und erst dann das Bild MOUNT.PIC. Die Information, daß zunächst *GraphiCraft* geladen werden soll, befindet sich in der Datei MOUNT.PIC.INFO. Ähnliches passiert auch mit BASIC-Programmen, die Sie mit AmigaBASIC erstellt und abgespeichert haben.

Es besteht ein Zusammenhang zwischen den in der Workbench verwendeten Icons beziehungsweise Schubladen-Icons und den Directories und Unterdirectories unter CLI. Das Aufrufen eines Directory entspricht dem Anklicken eines Schubladen-Icons. Geräte, die direkt auf gespeicherte Blöcke zugreifen können, wie die Diskettenlaufwerke (DF0:) oder eine Festplatte, werden ebenfalls über ein Icon, das Disketten-Icon, angesprochen.

Obwohl beide Schnittstellen, Workbench und CLI, mit AmigaDOS zusammenarbeiten, unterscheiden sie sich so weit voneinander, daß nicht alle Programme oder Anweisungen in beiden zugleich verwendet werden können. Die in Kapitel 2 dieses Handbuches beschriebenen Anweisungen können nur im CLI benutzt werden. Es gibt zum Beispiel zwei unterschiedliche DISKCOPY-Anweisungen. Die sich im Directory C befindende Anweisung wird von AmigaDOS im CLI gestartet, während die andere auf der Workbench aufgerufen wird.

Kapitel 1: AmigaDOS stellt sich vor

Dieses Kapitel gibt einen Überblick über das AmigaDOS-Betriebssystem, beschreibt die Handhabung des Terminals und der Hierarchie der Directories und zeigt den Gebrauch der DOS-Anweisungen. Am Ende des Kapitels finden Sie einfache Übungsbeispiele zum Einsatz der AmigaDOS-Anweisungen.

1.1 Überblick

AmigaDOS ist ein Multitasking-Betriebssystem, das speziell für den Amiga entwickelt wurde. *Multitasking* bedeutet, daß *gleichzeitig* mehrere Programme ablaufen oder daß mehrere Benutzer verschiedene Aufgaben erledigen können. Obwohl als Mehrplatz-Betriebssystem entwickelt, kann AmigaDOS in der Regel auch für den Einzelplatz-Einsatz gebraucht werden, wobei es dann mit seiner Multitasking-Fähigkeit Vordergrund- und Hintergrundprogramme simultan erledigt.

Jeder einzelne Arbeitsablauf des DOS-Betriebssystems, zum Beispiel die Dateiverwaltung, aber auch jedes laufende Programm wird Task oder Prozeß genannt (wir werden später sehen, daß es in Wirklichkeit einen Unterschied zwischen Task und Prozeß gibt). Es wird jeweils nur ein Prozeß zur selben Zeit ausgeführt. Während dieser Zeit warten andere Programme (das sind natürlich ebenfalls Prozesse) auf ihren Aufruf oder, nach einer Unterbrechung, auf ihre Fortsetzung. Jeder Prozeß erhält einen Rang (Priorität). Das Programm oder der Prozeß mit der jeweils höchsten Priorität wird sofort abgearbeitet, falls es nicht auf Eingaben wartet oder in ähnlicher Weise am Ablauf behindert ist. Prozesse mit niedrigerem Rang werden erst dann abgearbeitet, wenn ein Prozeß mit höherem Rang aus irgendeinem Grund angehalten wurde, beispielsweise zum Warten auf Daten, die von einer Diskette eingelesen werden sollen.

Das DOS des Amiga benutzt eine Reihe von Prozessen (oder Tasks), die dem normalen Anwender nicht zur Verfügung stehen, zum Beispiel für die Verwaltung der seriellen Datenübertragung. Diese Tasks werden auch *private Prozesse* genannt. Sie sind eigentlich lauter kleine eigenständige Programme. Andere private Prozesse überwachen das Terminal, die Dateiverwaltung auf der Diskette und den Betrieb von mehr als einem Diskettenlaufwerk.

Einer der Prozesse des AmigaDOS ist die Schnittstelle CLI. Sie können sowohl mehrere CLI-Fenster gleichzeitig öffnen und ihnen unterschiedliche Priorität zuweisen, als auch innerhalb eines CLI-Fensters mehrere CLI-Prozesse starten, die von 1 bis ... durchnumeriert werden. Sollen zusätzliche CLI-Prozesse zur Verfügung stehen, so können dafür die Befehle NEWCLI oder RUN verwendet werden. Ein CLI-Fenster schließen Sie mit der Anweisung ENDCLI. Die genannten Anweisungen werden in Kapitel 2 dieses Handbuches ausführlich behandelt.

1.2 Die Handhabung der Tastatur

Über die Tastatur (auch etwas fälschlich *Terminal* genannt) eingegebene Anweisungen können an das CLI gerichtet werden, das dem AmigaDOS mitteilt, zum Beispiel ein Programm zu laden. Anweisungen können aber auch an ein unter CLI laufendes Programm gerichtet werden. In diesem Fall führt ein *terminal handler* oder auch *console handler* die Ein- und Ausgabe durch. Diese Terminal-Kontrolle führt die aktuelle Zeilen-Editierung sowie verschiedene andere Funktionen aus. Sie können bis zu 255 Zeichen pro Zeile eingeben.

Mit der Taste <Backspace> können Sie eventuelle Tippfehler verbessern. Ein Druck auf diese Taste löscht das zuletzt eingegebene Zeichen. Geben Sie bei gedrückter Control-Taste (<Ctrl>) den Buchstaben »x« ein, so wird die Zeile, in der sich der Cursor befindet, gelöscht. Diese Control-Kombination werden wir im weiteren mit <Ctrl>-X bezeichnen.

Sobald irgend etwas eingegeben wird, stoppt AmigaDOS jede Ausgabe auf dem Bildschirm, bis die Eingabe (durch Betätigung der Taste <Return>) beendet ist, um eine verwirrende Mischung von Ein- und Ausgabe zu verhindern. AmigaDOS erkennt das Ende einer Anweisungszeile am Drücken der Return-Taste oder am Löschen einer Zeile mit <Ctrl>-X beziehungsweise <Backspace>, das so lange gedrückt bleiben muß, bis alle Zeichen der Zeile gelöscht sind. Sobald AmigaDOS erkannt hat, daß die Eingabe beendet wurde, wird die zurückgehaltene Ausgabe fortgesetzt. Die Ausgabe können Sie durch Drücken irgendeiner Taste (die Space-Taste dürfte am einfachsten sein) anhalten. Um die Ausgabe wieder aufzunehmen, müssen entweder die Backspace-, die Return-Taste oder <Ctrl>-X betätigt werden.

Die Tastenkombination <Ctrl>-/ erkennt das AmigaDOS als Dateiende-Zeichen. In verschiedenen Fällen wird durch diese Kombination eine Eingabedatei beendet. Diese Anweisungen werden in Abschnitt 1.3.6 genauer behandelt.

Wenn fremdartige Zeichen auf dem Bildschirm erscheinen, ohne daß Sie etwas auf der Tastatur eingegeben haben, haben Sie sicher versehentlich <Ctrl>-O gedrückt. Das DOS des Amiga sieht diese Kombination als eine Anweisung für das *Console-Device* (CON:) an, den alternativen Zeichensatz zu verwenden. Dies können Sie durch <Ctrl>-N wieder rückgängig machen. Alle bisherigen Zeichen sollten nun wieder normal erscheinen. Es ist auch möglich, den Bildschirm mit <Esc>-C zu löschen und wieder normalen Text ausgeben zu lassen.

Achtung: Bei der Eingabe über das Console-Device CON: funktionieren weder die Funktions- noch die Cursor-Tasten. Sollen diese Tasten wieder verwendet werden können, so muß das interne Device RAW: verwendet werden. Eine Beschreibung des RAW:-Device kann im Abschnitt *1.3.6 Zum Verständnis von Device-Namen* in diesem Kapitel nachgeschlagen werden.

AmigaDOS erkennt alle eingegebenen Anweisungen und Argumente in Groß- und/oder Kleinschrift. Das DOS des Amiga zeigt einen Dateinamen in der Form an, in der er beim Speichern der Datei gebildet wurde. Im Gegensatz dazu wird er vom DOS in jeder Form gefunden, unabhängig, welche Schriftart Sie zum Bezeichnen des Dateinamens bei erneuter Öffnung verwenden.

1.3 Alles über Files

Dieser Abschnitt beschreibt die AmigaDOS-Dateiverwaltung. Im einzelnen erklärt er, wie Dateien benannt, organisiert und aufgerufen werden.

Eine *Datei* beziehungsweise ein *File* ist die kleinste benannte Dateneinheit, die von AmigaDOS benutzt wird. Die einfachste Identifikation einer Datei wird durch die Angabe des Filenamens erreicht, wie in Abschnitt 1.3.1 beschrieben wird. Unter Umständen kann es nötig sein, ein File genauer zu benennen. Diese Benennung enthält dann den Device- oder Diskettennamen und/oder Directory-Namen zusätzlich zum Filenamen. Auch dies wird in den nächsten Abschnitten besprochen.

1.3.1 Die Filenamen

Das DOS des Amiga behält Daten und Informationen in einer Anzahl von Dateien auf einer Diskette. Sie werden so benannt, daß sie leicht wiedererkannt und aufgerufen werden können. Die Dateienverwaltung erlaubt Dateinamen, die bis zu 30 Zeichen lang sein können. Alle verfügbaren Zeichen des Amiga, außer Doppelpunkt (:) und Schrägstrich (/), sind zulässig. Das heißt, in einem Dateinamen dürfen Leerzeichen, Gleichheitszeichen, das Plus (+) und Anführungsstriche (") vorkommen (auch mehrere Punkte sind erlaubt), also alle Zeichen, die von CLI anerkannt werden. Nichtsdestotrotz müssen Sie den Dateinamen in Anführungszeichen angeben, wenn Sie diese speziellen Zeichen (Leerzeichen,

Anführungsstrich, Schrägstrich) in einem Dateinamen verwenden. Sollen dagegen Anführungsstriche im Dateinamen verwendet werden, so ist unmittelbar davor ein »Sternchen« (*) zu setzen. Ähnlich muß zur Verwendung eines Sternchens zusätzlich ein zweites Sternchen eingefügt werden. Das bedeutet, daß Sie den Dateinamen

```
A*B = C"
```

wie folgt eingeben müssen:

```
"A**B = C*""
```

damit er von dem CLI angenommen wird.

Vorsicht! Dieser Gebrauch des Sternchens in dieser Weise widerspricht vielen anderen Betriebssystemen, wo es allgemein als *Joker* verwendet wird. Dieser Joker bewirkt, daß bei Eingabe von DAT* alle Dateien, die mit diesen drei Zeichen beginnen, ausgewählt werden. Wird unter AmigaDOS ein Sternchen ohne weitere Zeichen eingegeben, so repräsentiert es die Tastatur und das aktuelle Fenster.

Zum Beispiel kopiert

```
COPY Name TO *
```

die Datei *Name* auf den Bildschirm.

Leerzeichen sollten Sie am Anfang und am Ende eines Dateinamens vermeiden, da sie leicht zu Mißverständnissen führen.

1.3.2 Directories anlegen

In der Dateiverwaltung werden Directories zum Zusammenfassen von Dateien in logischen Einheiten gebraucht. So können Sie zum Beispiel zwei verschiedene Directories dazu verwenden, um Programme von Programmerläuterungen zu trennen. Oder um Dateien einer Gruppe von denen anderer Gruppen zu unterscheiden.

Jede Datei auf einer Diskette muß in einem Directory stehen. Eine leere Diskette enthält ein Directory, das als Quelldirectory oder auch Hauptinhaltsverzeichnis bezeichnet wird. Wenn eine Datei auf eine leere Diskette geschrieben wird, dann steht sie in diesem Quelldirectory. Directories können beliebig viele weitere Directories beinhalten. In jedem Directory können deshalb Dateien, weitere Directories oder beides stehen. Jeder Dateiname ist innerhalb des Directories, in dem er steht, einzigartig. Andererseits dürfen zwei Dateien in unterschiedlichen Directories denselben Namen annehmen. Auf diese Weise unterscheidet sich die Datei MARTIN im Directory THOMAS vollkommen von einer Datei mit dem Namen MARTIN im Directory CAROLA.

Teilen sich zwei Anwender oder zwei Prozesse eine Diskette, so brauchen sie nicht zu befürchten, daß sie sich gegenseitig die Dateien überschreiben, solange sie in ihrem eigenen Directory bleiben.

> **Vorsicht!** Soll eine Datei erzeugt werden, deren Dateiname bereits existiert, löscht AmigaDOS den eventuell bestehenden Inhalt der Datei. Falls eine Datei gelöscht wird, wird darüber keine Meldung auf dem Bildschirm ausgegeben.

Der Aufbau der Directories kann ebenso zur Organisation von Informationen auf der Diskette verwendet werden, zum Beispiel, um verschiedene Arten von Dateien in verschiedenen Directories aufzubewahren.

Dazu ein Beispiel: Angenommen, eine Diskette enthält zwei Directories mit Namen BERND und MONIKA. Das Directory BERND enthält zwei Dateien, die TEXT und BRIEF heißen. Das Directory MONIKA enthält eine Datei mit Namen DATEN und zwei Directories mit Namen BRIEF und RECHNUNG. Diese Unterdirectories enthalten je eine Datei JUNI18. Bild 1.1 stellt den Aufbau des Beispiels dar:

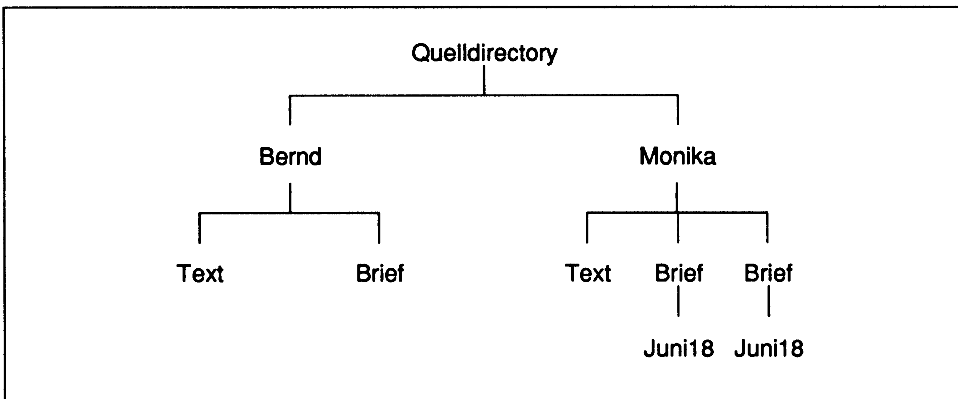


***Bild 1.1:** Aufbau einer Directory-Struktur*

Achtung: Das Directory BERND hat eine Datei mit Namen BRIEF, während das Directory MONIKA ein Directory mit Namen BRIEF besitzt. Hier kann keine Verwirrung entstehen, weil beide Dateien sich in verschiedenen Directories befinden. Die Verzweigung der Directories kann nach unten beliebig fortgeführt werden.

Um eine Datei vollständig zu bezeichnen, muß in den Dateinamen das Directory, in dem die Datei sich befindet, bis zum Quelldirectory einbezogen sein. Um eine Datei zu bezeichnen, geben Sie die Namen der Directories auf dem Weg zur angestrebten Datei an. Um jeden Directory-Namen vom nächsten Namen zu trennen, muß ein Schrägstrich (/) verwendet werden. Die vollständigen Bezeichnungen der Datendateien in Bild 1.1 lauten dann wie folgt:

```
Bernd/Text
Bernd/Brief
Monika/Daten
Monika/Brief/Juni18
Monika/Rechnung/Juni18
```

1.3.3 Setzen des aktuellen Directory

Es ist sehr aufwendig, eine vollständige Dateibezeichnung, wie oben beschrieben, einzugeben. Aus diesem Grunde stellt die Dateienverwaltung Ihres Amiga ein *aktuelles Directory* als Bezugspunkt zur Verfügung. Die Dateiverwaltung sucht in diesem aktuellen Directory nach der benötigten Datei. Um das aktuelle Directory festzulegen, verwenden Sie den Befehl CD, die Abkürzung für *Current Directory*. Wird in dem obigen Beispiel das Directory MONIKA mit der Befehlssequenz

```
CD Monika
```

zum aktuellen Directory gemacht, dann reichen die folgenden Namen zum Bezeichnen der Dateien aus:

```
Daten
Brief/Juni18
Rechnung/Juni18
```

Jedes Directory kann zum aktuellen Directory gemacht werden. Um irgendeine Datei innerhalb dieses Directory zu bezeichnen, genügt es also, den Namen der Datei einzugeben. Um Dateien in Unter-Directories des aktuellen Directory zu bestimmen, genügt es, die Namen der Directories vom aktuellen Directory ausgehend abwärts zu benennen.

Alle anderen Dateien auf der Diskette sind noch wie üblich erreichbar, auch wenn jetzt ein aktuelles Directory gesetzt ist. Um AmigaDOS allerdings alle Directories vom Quelldirectory aus durchsuchen zu lassen, setzen Sie einen Doppelpunkt (:) an den Anfang des Dateinames. Wird MONIKA zum aktuellen Directory, kann die Datei DATEN immer noch durch Eingabe der Bezeichnung :MONIKA/DATEN erreicht werden. Das aktuelle Directory erspart auf einfache Art eine Menge Tipparbeit, weil nun statt dessen der Dateiname DATEN ausreicht.

Die anderen Dateien auf der Diskette erreichen Sie mit :BERND/TEXT beziehungsweise :BERND/BRIEF, mit der Anweisung CD oder durch Eingabe von »/« vor dem Dateinamen. Ein Schrägstrich bedeutet nicht *Ursprung* (*root*) wie in einigen Systemen. Er verweist

vielmehr auf das Directory über dem aktuellen Directory. AmigaDOS erlaubt die mehrfache Verwendung von Schrägstrichen. Jeder Schrägstrich verweist auf eine Stufe höher. So ist ein UNIX »/« ein »/« im AmigaDOS. In ähnlicher Weise ist ein MS-DOS »\« in AmigaDOS ein »/«. Ist zum Beispiel :MONIKA/BRIEF das aktuelle Directory, können Sie die Datei :MONIKA/RECHNUNG/JUNI18 als /RECHNUNG/JUNI18 bezeichnen. Um die Dateien in :BERND zu erreichen, geben Sie zuerst ein:

```
CD :Bernd
```

oder

```
CD //Bernd
```

Danach können Sie jede Datei in BERND einfach nur mit dem Dateinamen bezeichnen. Natürlich dürfen Sie auch immer »//« zum direkten Zugriff auf die spezielle Datei benutzen. Zum Beispiel:

```
TYPE //Bernd/Brief
```

Dieser Ausdruck zeigt die Datei, ohne daß zuvor BERND als aktuelles Directory gesetzt werden muß. Um direkt auf die Stufe des Quelldirectorys zu gelangen, müssen Sie nur einen Doppelpunkt (:) vor dem Directory-Namen eingeben. Mit jedem Schrägstrich gelangt man nur eine Stufe zurück.

1.3.4 Benennung des aktuellen Device

Ein Device ist ein Peripheriegerät, wie Diskettenlaufwerk, Drucker, Festplatte, Tastatur oder Bildschirm. Stehen Ihnen mehrere Diskettenlaufwerke zur Verfügung, wird jede Floppy in der Form DFn bezeichnet. Die Variable »n« steht dabei für die Nummer des Laufwerks, zum Beispiel entspricht DF1 dem zweiten angeschlossenen Diskettenlaufwerk. Diese Bezeichnung des Diskettenlaufwerks ist unser erstes Beispiel für einen Device-Namen. Augenblicklich kann AmigaDOS mit maximal vier Diskettenlaufwerken zusammenarbeiten. Das heißt, es sind die Device-Namen DF0 bis DF3 erlaubt. Jede Diskette sollte mit einem möglichst einmaligen Namen bezeichnet werden; die Diskettenbezeichnung wird weiter unten detaillierter beschrieben.

Ferner existiert ein logisches Device mit dem klangvollen Namen SYS:. Es bezieht sich auf die Diskette, von welcher das System gestartet wurde. Diese Gerätebezeichnung läßt sich auch statt des Device-Namens eines Diskettenlaufwerks, zum Beispiel DF0:, verwenden.

Das aktuelle Directory bezieht sich stets auf ein aktuelles Laufwerk, das Laufwerk, auf dessen Diskette sich das Directory befindet. Wie Sie wissen, wird das Quelldirectory des aktuellen Laufwerks durch einen Doppelpunkt vor der Dateibeschreibung gekennzeichnet. Das Quelldirectory eines speziellen Laufwerks wird angegeben, indem der Doppelpunkt nach der Laufwerksbezeichnung eingetippt wird. Dies ist eine weitere Möglichkeit, die Datei DATEN im Directory MONIKA zu bezeichnen: DF1:MONIKA/DATEN. Sie setzt

natürlich voraus, daß die Diskette im Laufwerk DF1 liegt. Um auf eine Datei, beispielsweise bezeichnet als PROJEKTBERICHT, im Directory PETER des Laufwerkes DF0 zu verweisen, muß DF0:PETER/PROJEKTBERICHT eingegeben werden. Dabei ist es bei dieser Schreibweise unwichtig, welches Directory vorher zum aktuellen bestimmt wurde.

Wenn Sie auf ein Diskettenlaufwerk oder irgendein anderes Device verweisen, ob mit oder ohne Directory-Namen, müssen Sie stets den Doppelpunkt eingeben, wie zum Beispiel DF1:.

Tabelle 1.1 illustriert noch einmal den Aufbau einer Datei-Bezeichnung. Danach finden Sie einige Beispiele gültiger Dateinamen.

Links eines :	Rechts eines :	Rechts eines /
Device-Name oder Diskettenname	Directory-Name oder Dateiname	Subdirectory-Name oder Dateiname

***Tabelle 1.1:** Aufbau einer Dateibezeichnung*

```
SYS:Anweisungen
DF1:Monika/Brief
DF2:Monika/Brief/Jun18
DOC:Bericht/Teil1/Abbildungen
FONTS:Silly-Font
C:cls
```

Statt des Device-Namens dürfen Sie auch den Diskettennamen zum Aufrufen einer einzelnen Diskette verwenden. Für den Fall, daß sich die gesuchte Datei auf einer Diskette mit dem Namen MCC befindet, wird diese Datei durch Eingabe des vollständigen Namens MCC:PETER/PROJEKTBERICHT bezeichnet. Mit dem Diskettennamen sprechen Sie eine Diskette an, unabhängig davon, in welchem Laufwerk sie sich befindet, unabhängig sogar davon, ob sie sich überhaupt in einem Laufwerk befindet. Den Diskettennamen geben Sie Ihrer Diskette beim Formatieren. Weitere Einzelheiten dazu finden sich unter der Anweisung FORMAT in Kapitel 2 dieses Handbuches.

Ein Device-Name ist im Gegensatz zum Diskettennamen nicht wirklich ein Bestandteil der benötigten Bezeichnung. Haben Sie beispielsweise eine Diskette im Laufwerk DF0: erstellt, so kann sie vom AmigaDOS auch in einem anderen Laufwerk, zum Beispiel DF1:, gelesen werden. Vorausgesetzt, die beiden Laufwerke besitzen das gleiche Diskettenformat (3,5-Zoll- oder 5,25-Zoll-Format) und sind untereinander austauschbar. Wurde eine Datei BERND auf einer Diskette in Laufwerk DF0: erzeugt, lautet ihre Bezeichnung DF0:BERND. Wird dann

die Diskette mit der Datei BERND in Laufwerk DF1: eingelegt, kann das AmigaDOS diese Datei mit DF1:BERND lesen.

1.3.5 Dateinotizen anhängen

Obwohl schon der Dateiname Auskunft über deren Inhalt geben kann, ist es oft notwendig, die Datei näher zu untersuchen, um weitere Einzelheiten herauszufinden. AmigaDOS bietet eine einfache Lösung dieses Problems. Der Befehl FILENOTE hängt an eine Datei einen Kommentar an. Solche Kommentare dürfen aus bis zu 80 Zeichen bestehen. Kommentare, die Leerzeichen enthalten, müssen zwischen Anführungszeichen (") stehen. Der Kommentar kann zum Beispiel die Versionsnummer eines Programms oder einen im Programm versteckten Fehler beschreiben und vieles mehr.

Ein Kommentar ist an eine konkrete Datei gebunden. Wenn Sie eine Datei eröffnen, hat diese noch keinen Kommentar. Wird eine Datei kopiert, enthält die Kopie ebenfalls keinen Kommentar, wird sie jedoch nur überschrieben, bleibt der Kommentar erhalten. Eine umbenannte Datei behält ebenfalls ihren Kommentar. Weitere Einzelheiten über Dateinotizen finden Sie in Kapitel 2 unter den Anweisungen LIST, RENAME und FILENOTE.

1.3.6 Zum Verständnis von Device-Namen

Devices haben Namen, damit Sie unter deren Angabe auf sie verweisen können. Laufwerksbezeichnungen wie DF0: oder DF1: sind Beispiele für Device-Namen. Wie bei den Dateinamen können Device-Namen in Groß- und Kleinschrift eingegeben werden. Bei Disketten muß dem Device-Namen eine Dateibezeichnung folgen, da AmigaDOS Dateien auf diesen Devices unterhält. Weiterhin kann die Dateibezeichnung Directory-Namen beinhalten, da Directories ebenfalls von AmigaDOS unterhalten werden.

Sie sind in der Lage, im Speicher Dateien zu erzeugen, die Sie mit dem Device-Namen RAM: bezeichnen. RAM: führt im Speicher eine Dateienverwaltung aus, die alle Anweisungen der üblichen Dateienverwaltung verstehen und nachvollziehen kann. Wir sprechen bei diesem Device auch von einer RAM-Disk.

Achtung! RAM: benötigt auf Diskette die Bibliothek L/RAM-HANDLER. Das Device RAM: schreibt jeweils in Blöcken zu je 512 Byte in den Speicher, paßt also den verbrauchten Speicherplatz stets dem tatsächlich benötigten an (dynamische RAM-Disk). Falls Sie dieses Device bei jedem Neustarten Ihres Rechners installieren möchten, so brauchen Sie nur das Startup-File zu ändern.

Sobald ein Device RAM: existiert, können Sie bei Bedarf ein Directory erzeugen, um alle AmigaDOS-Befehle in den Speicher zu kopieren. Dies erreicht man durch folgende Eingaben:

```
MAKEDIR RAM:C
COPY SYS:C RAM:C ALL
ASSIGN C: RAM:C
```

Das Ergebnis kann mit DIR RAM: sichtbar gemacht werden. In unserem Beispiel würde das Inhaltsverzeichnis das Directory C beinhalten. Der Befehl DIR listet dies als C(DIR) auf. Das Speichern der Anweisungen im Device RAM: beschleunigt das Laden der Anweisungen sehr, hat aber den Nachteil, daß im Speicher weniger Platz für alles andere übrigbleibt (schließlich wird ja RAM verbraucht). Wird der Amiga ausgeschaltet oder mit »Reset« zurückgesetzt, geht jede Datei im Device RAM: verloren.

AmigaDOS bietet weitere Devices an, die Sie statt eines Bezugs auf eine Diskettendatei verwenden können. Die folgenden Absätze beschreiben diese Devices, zu denen SER:, PAR:, PRT:, CON: und RAW: gehören.

Das Device SER: verweist auf ein Gerät, das an den seriellen Port angeschlossen ist. In den meisten Fällen ist dies ein Drucker. Wenn folgende Sequenz eingegeben wird:

```
COPY xyz TO ser:
```

wird AmigaDOS angewiesen, den Inhalt der Datei XYZ (zum Beispiel einen Text) seriell zu übertragen. Es ist zu beachten, daß das serielle Device zur gleichen Zeit nur in Vielfachen von 400 Byte kopiert. Eine Kopie mit SER: kann deshalb sehr grobkörnig erscheinen.

Das Device PAR: verweist ähnlich dem Device SER: auf den parallelen Port.

AmigaDOS stellt ebenso das Device PRT: zur Verfügung, hergeleitet von *PRinTer*, zu deutsch *Drucker*. PRT: ist der in PREFERENCES festgelegte Drucker. In diesem Programm können Sie dem Amiga mitteilen, ob Ihr Drucker am seriellen oder am parallelen Port angeschlossen ist (PRT: bezieht sich also entweder auf den seriellen oder auf den parallelen Port). Die Befehlssequenz lautet:

```
COPY xyz TO prt:
```

Sie druckt die Datei XYZ unabhängig davon, ob der Drucker am seriellen oder am parallelen Port angeschlossen ist.

PRT: verwandelt jedes Zeilenvorschub-Zeichen in einer Datei in einen Wagenrücklauf mit Zeilenvorschub. Einige Drucker benötigen diese zusätzliche Übersetzung der Datei nicht. Sollen deshalb in einer Datei die Zeilenvorschub-Zeichen nur als solche ohne Wagenrücklauf gesendet werden, so verwenden Sie PRT:RAW anstatt PRT:.

AmigaDOS unterstützt mehrere Bildschirmfenster. Um ein neues Fenster zu erzeugen, können Sie das Device CON: zu Hilfe ziehen. Das Format für CON: lautet wie folgt:

```
CON:X/Y/Breite/Höhe/[Name]
```

dabei sind X und Y die Fensterkoordinaten; *Breite* und *Höhe* die ganzzahligen Variablen der Fensterbreite und -höhe; und optional ein Name in der Form einer Zeichenkette. Der Name erscheint in der Kopfzeile des Fensters. Die Schrägstriche (/) müssen mit eingegeben werden, einschließlich des letzten, nach der Höhenangabe, auch wenn kein Name mehr folgt. Der Name kann bis zu dreißig Zeichen lang sein, einschließlich Leerzeichen. Sollten in dem Namen Leerzeichen vorkommen, so muß die ganze Anweisung zwischen Anführungsstrichen (") stehen, wie in folgendem Beispiel:

```
"CON:20/10/300/100/Fenster3.0"
```

Es gibt ein weiteres Fenster-Device, das mit dem Namen RAW: beginnt, aber für den normalen Anwender wenig Nutzen hat. In Kapitel 2 des *AmigaDOS-Programmierer-Handbuchs* finden Sie dazu weitere Einzelheiten. RAW: wird ähnlich wie CON: zum Erzeugen eines unbearbeiteten Fensters gebraucht. Aber im Gegensatz zu CON: übersetzt RAW: keine Zeichen und ermöglicht keine Änderung des Inhalts einer Zeile. Wichtig ist, daß RAW: Eingaben annimmt und sie genau in der Form wieder ausgibt, in der sie zuerst eingetippt wurden. Das bedeutet, daß die Zeichen unmittelbar in ein Programm übertragen werden, ohne daß sie mit der Backspace-Taste gelöscht werden können. Normalerweise findet RAW: in einem Programm Verwendung, in dem Ein- und Ausgaben ohne Veränderung der Zeichen gewünscht werden.

> **Achtung!** RAW: ist für den fortgeschrittenen Anwender gedacht. Mit RAW: sollte nicht experimentiert werden.

Es gibt ein spezielles Zeichen, das Sternchen (*), das bei Ein- wie bei Ausgaben auf das aktuelle Fenster verweist. Der Befehl COPY kann zum Kopieren von Datei zu Datei verwendet werden. Zum Beispiel kann mit dem Sternchen vom aktuellen Fenster in ein neugeschaffenes Fenster kopiert werden (das hat natürlich meist nur dann Sinn, wenn die Datei ASCII-Text enthält):

```
COPY * TO CON:20/20/350/150/
```

Oder von einer Datei ins aktuelle Fenster kopieren:

```
COPY Bernd/Brief TO *
```

AmigaDOS beendet den Kopiervorgang, sobald es das Dateiende erreicht. Um dem AmigaDOS mitzuteilen, daß es mit Kopieren aufhören soll, müssen Sie die Kombination <Ctrl>-\ eingeben.

Beachten Sie bitte, daß * kein allgemeingültiger Joker ist.

1.3.7 Vereinbarungen zu Directories und logischen Devices

Zusätzlich zu den vorher erwähnten physikalischen Devices bietet AmigaDOS eine Reihe nützlicher logischer Devices an. Das DOS benutzt diese Devices, um Dateien zu finden, die Programme von Zeit zu Zeit brauchen. Auf diese Weise kann ein Programm auf ein normales Device weisen, unabhängig davon, wo die Datei sich momentan befindet. Diese ganzen logischen Devices können von Ihnen neu auf ein bestimmtes Directory gerichtet (engl.: assign) werden.

Die logischen Devices werden in diesem Abschnitt wie folgt bezeichnet:

Name	Beschreibung	Directory
SYS:	Quelldirectory der Systemdiskette	:
C:	Anweisungsübersicht	:C
L:	Directory der Systembibliothek	:L
S:	Sequenzübersicht	:S
LIBS:	Bibliothek der geöffneten Übersichtsaufrufe	:LIBS
DEVS:	Device der geöffneten Deviceaufrufe	:DEVS
FONTS:	Ladbare Zeichensätze	:FONTS
	Platz für temporäre Dateien	:T

Tabelle 1.2: *Logische Devices*

Logischer Device-Name: SYS:
Typischer Directory-Name: Start.Disk:

SYS: steht für das Quelldirectory der SYStem-Diskette. Wenn das Amiga-System gestartet wird, wird das SYS: dem Quelldirectory auf der Diskette in Laufwerk DF0: durch das DOS zugewiesen. Für den Fall, daß die Diskette in DF0: den Diskettennamen START.DISK: hat, weist AmigaDOS SYS: an START.DISK:. Nach dieser Zuweisung benutzt jedes Programm, das auf SYS: verweist, dessen Quelldirectory.

Logischer Device-Name: C:
Typischer Directory-Name: Start.Disk:c

C steht für das Befehlsdirectory. Wenn Sie eine Anweisung mit CLI eingeben, zum Beispiel DIR, sucht AmigaDOS nach der Anweisung zuerst im aktuellen Directory. Wenn das Betriebssystem den Befehl im aktuellen Directory nicht findet, sucht es nach C:DIR. Wurde C: einem anderen Directory zugeteilt, zum Beispiel RAM:C, dann liest und führt AmigaDOS Befehle von RAM:C/DIR aus.

Logischer Device-Name: L:
Typischer Directory-Name: Start.Disk:l

L steht für das Bibliotheks-Directory. Dieses Directory enthält die genauen Arbeitsanweisungen langer Befehle und nicht festeingebundene Teile des Betriebssystems, die auf Diskette stehenden Echtzeitbibliotheken, wie RAM-HANDLER, PORT-HANDLER, DISK-VALIDATOR und so weiter, werden hier aufbewahrt. AmigaDOS braucht dieses Directory!

Logischer Device-Name: S:
Typischer Directory-Name: Start.Disk:S

S steht für die Sequenz-Übersicht. Sequenzdateien enthalten Anweisungssequenzen, nach denen der Befehl EXECUTE sucht und die er benutzt. EXECUTE sucht zuerst nach der Sequenzdatei im aktuellen Directory. Wenn EXECUTE sie dort nicht finden kann, sucht er in dem Directory, auf das mit S: verwiesen wird.

Logischer Device-Name: LIBS:
Typischer Directory-Name: Start.Disk:LIBS

Open Library-Funktionsaufrufe suchen hier nach der gesuchten Bibliothek, falls sie nicht bereits in den Speicher geladen wurde.

Logischer Device-Name: DEVS:
Typischer Directory-Name: Start.Disk:DEVS

Open Device-Funktionsaufrufe suchen hier nach dem Device, falls es nicht bereits in den Speicher geladen wurde.

Logischer Device-Name: FONTS:
Typischer Directory-Name: Start.Disk:FONTS

Open Fonts-Funktionsaufrufe suchen hier nach ladbaren Zeichen, falls sie nicht bereits in den Speicher geladen wurden.

Zusätzlich zu den oben genannten zuteilbaren Directories eröffnen viele Programme Dateien in dem Directory :T. Wie Sie sich erinnern werden, wird im Quelldirectory einem Datei- oder Directory-Namen ein Doppelpunkt (:) vorangestellt. Deshalb ist :T das Directory T im Quelldirectory auf der aktuellen Diskette. Dieses Directory wird zum Speichern aktueller Dateien verwendet. Programme, ebenso wie Editoren, legen in diesem Directory ihre gegenwärtigen Arbeitsdateien oder Sicherheitskopien der zuletzt bearbeiteten Datei ab. Sollte der Speicherplatz auf der Diskette knapp werden, so ist hier die Suche nach Dateien, die nicht länger benötigt werden, einer der ersten Schritte zur Lösung des Problems.

Wenn das Betriebssystem zum ersten Mal geladen wird, weist AmigaDOS zuerst C: dem Directory :C zu. Angenommen, Sie laden eine Diskette, die Sie mit dem folgenden Befehl formatiert haben:

```
FORMAT DRIVE DF0 NAME Start.Disk
```

In diesem Fall wird dem SYS: das Directory START.DISK zugeteilt. Das *logische Device* C: wird auf der gleichen Diskette, als START.DISK:C, dem Directory C zugewiesen. Zugleich sind folgende Zuteilungen nach dem Starten des Rechners erledigt:

```
C:          Start.Disk:c
L:          Start.Disk:l
S:          Start.Disk:s
LIBS:       Start.Disk:libs
DEVS:       Start.Disk:devs
FONTS:      Start.Disk:fonts
```

Wenn ein Directory nicht ansprechbar ist, wird dem Quelldirectory ein entsprechendes logisches Device zugeteilt.

Glückliche Besitzer einer Harddisk (DH0:) müssen dem Betriebssystem folgende Anweisungen geben, wenn sie die Systemdateien auf ihr nutzen wollen:

```
ASSIGN SYS:             DH0:
ASSIGN C:               DH0:C
ASSIGN L:               DH0:L
ASSIGN S:               DH0:S
ASSIGN LIBS:            DH0:LIBS
ASSIGN DEVS:            DH0:DEVS
ASSIGN FONTS:           DH0:FONTS
```

Wichtig ist, daß solche Zuweisungen für alle CLI-Prozesse gelten. Wenn innerhalb eines Fensters eine Zuteilung geändert wird, ändert sie sich auch für alle anderen Fenster. Soll mit einer selbsterstellten Zeichensatzbibliothek gearbeitet werden, und der Zeichensatz ist auf der Diskette ZEICHENSATZ DISK unter SELBSTFONTS gespeichert, muß folgende Anweisung eingegeben werden:

```
ASSIGN FONTS: "Zeichensatz Disk:eigenefonts"
```

Sollen Anweisungen schneller ausgeführt und kann dabei ruhig Speicherplatz *verheizt* werden, dann ist folgende Anweisungssequenz richtig:

```
MAKEDIR RAM:C
COPY SYS:C RAM:C ALL
ASSIGN C: RAM:C
```

Dadurch werden alle normalen Befehle des AmigaDOS in eine RAM-Disk geladen und die Zuteilung des Befehlsdirectory geändert, so daß sie in der RAM-Disk gefunden werden.

1.4 Theorie und Praxis der AmigaDOS-Befehle

Eine Anweisung des AmigaDOS besteht aus dem Anweisungsbegriff und seinen Argumenten, falls welche benötigt werden. Damit eine Anweisung des AmigaDOS ausgeführt wird, muß der Befehl sofort hinter dem CLI-Zeichen eingegeben werden. Dieses CLI-Zeichen besteht aus einer Zahl n und dem Zeichen »>«. Die Variable n steht für die laufende Nummer des jeweiligen CLI-Prozesses. Dieses Zeichen, auch *Prompt* genannt, können Sie mit der Anweisung PROMPT, wie in Kapitel 2 beschrieben, ändern.

Wird eine Anweisung eingegeben, arbeitet sie als ein Teil des CLI. Man kann nach einer Anweisung weitere eingeben, sie werden aber erst ausgeführt, sobald der momentan bearbeitete Befehl erledigt ist. Wenn eine Anweisung ausgeführt wurde, erscheint wieder das CLI-Zeichen. In diesem Fall arbeitet der Befehl interaktiv, das heißt im Dialog mit dem Anwender.

Achtung! Wird mit einem Befehl interaktiv gearbeitet und kann er nicht ausgeführt werden, so geht AmigaDOS zur nächsten eingegebenen Anweisung weiter und arbeitet diese ab. Es ist also gefährlich, zu viele Befehle hintereinander einzugeben. Wenn zum Beispiel

```
COPY a TO b
DELETE a
```

eingegeben wurde und der COPY-Befehl nicht ausgeführt werden kann, zum Beispiel, weil die Diskette voll ist, wird DELETE abgearbeitet, und die Datei *a* ist verloren.

1.4.1 Befehle im Hintergrund ausführen lassen

Man kann AmigaDOS beauftragen, eine oder mehrere Anweisungen im Hintergrund abzuarbeiten. Dazu wird der Befehl RUN verwendet. Er erzeugt einen neuen CLI-Prozeß niedrigerer Priorität. In diesem Fall werden die nachfolgenden Befehlszeilen simultan zu den mit RUN gestarteten ausgeführt. Zum Beispiel können Sie den Inhalt eines Directory zur gleichen Zeit auslesen, zu der eine Textdatei ausgedruckt wird. Dies erreicht man mit:

```
RUN TYPE textdatei TO PRT:
LIST
```

RUN erzeugt ein neues CLI-Ablaufprogramm und führt den Ausdruck durch, während im ursprünglichen CLI-Fenster die Dateien des Directory aufgelistet werden.

Mit RUN kann AmigaDOS verschiedene Anweisungen ausführen. RUN arbeitet jeden Befehl in der gegebenen Reihenfolge ab. Der Rest der Zeile hinter RUN Anweisungen

enthält Anweisungen. Beenden Sie die Befehlszeile durch Drücken der Return-Taste. Um eine Anweisungszeile um weitere Zeilen zu erweitern, wird ein Plus-Zeichen (+) vor dem Drücken der Return-Taste eingegeben. Es können beliebig viele Anweisungszeilen verkettet werden. Beispiel:

```
RUN JOIN textdatei1 textdatei2 AS textdatei+
SORT textdatei TO sortiertext +
TYPE sortierttext TO PRT:
```

1.4.2 Handhabung der EXECUTE-Kommando-Files

Der Befehl EXECUTE kann dazu verwendet werden, in einer Datei gespeicherte Anweisungen abzuarbeiten, anstatt sie im Direktmodus einzugeben. Das CLI liest aus der Datei eine Reihe von Anweisungen, bis es auf einen Fehler oder das Ende der Datei stößt. Wird ein Fehler gefunden, so werden nachfolgende Befehle in der Zeile mit RUN oder in der EXECUTE-Datei nicht ausgeführt, es sei denn, der Befehl FAILAT wurde benützt. In Kapitel 2 dieses Handbuchs finden Sie eine ausführliche Beschreibung des Befehls FAILAT. Das CLI liefert nur ein Prompt, nachdem interaktiv ausgeführte Befehle abgearbeitet sind.

1.4.3 Ein- und Ausgaberichtungen von Befehlen

AmigaDOS ermöglicht die Änderung der normalen Eingabe- und Ausgabe-Richtung. Die Symbole Größer als (>) und Kleiner als (<) werden als Zeichen verwendet. Wird ein Befehl eingegeben, zeigt AmigaDOS das Ergebnis des Befehls auf dem Bildschirm an. Um die Ausgabe in eine Datei schreiben zu lassen, kann das »>« verwendet werden. Mit dem »<« teilen Sie dem AmigaDOS mit, daß es die Eingabe aus einer genauer bezeichneten Datei in ein Programm lädt, anstatt sie von der Tastatur zu übernehmen. Die Zeichen »>« und »<« arbeiten wie Verkehrspolizisten, die den Informationsfluß lenken. Um zum Beispiel die Ausgabe von DATE in eine Datei TEXTDATEI umzuleiten, müssen Sie folgenden Befehl eingeben:

```
DATE > textdatei
```

In Kapitel 2 dieses Handbuchs ist eine vollständige Erläuterung der Symbole < und > zu finden.

1.4.4 Unterbrechungen in AmigaDOS

AmigaDOS bietet vier verschiedene Anweisungen, um Abläufe abzubrechen: CTRL-C, CTRL-D, CTRL-E und CTRL-F. Um den momentan arbeitenden Befehl zu beenden, drücken Sie die Kombination CTRL-C. In einigen Fällen, z.B. bei EDIT, bewirkt das Drücken von CTRL-C nur eine Unterbrechung und ein weiteres Einlesen von EDIT-Befehlen. Wollen Sie dem CLI mitteilen, daß eine mit EXECUTE initialisierte Befehlsfolge bei Ende des momentan arbeitenden Befehls aufzuhören hat, drücken Sie die Kombination

CTRL-D. CTRL-E und CRTL-F werden nur in speziellen Fällen von einigen Befehlen verwendet. Genaueres ist im *AmigaDOS-Programmierer-Handbuch* nachzulesen.

Beachten Sie bitte, daß diese Abbruch-Flags vom laufenden Programm abgefragt werden müssen und dieses daraufhin seine Arbeit einstellen muß. AmigaDOS selbst wird nie ein Programm zerstören.

1.4.5 Ein Wort zum Befehlsformat

Dieser Abschnitt erklärt die von allen Anweisungen des AmigaDOS verwendete Form und Anordnung der Argumente. Die Zahl und Form der Argumente wird durch Schlüsselwörter bezeichnet. Einige der Schlüsselworte können synonym verwendet werden. In diesem Fall sind sie durch ein »ist gleich« (=) gekennzeichnet. Die folgende Form spezifiziert drei Argumente:

```
ABC, WWW, XYZ = ZZZ
```

Dabei können die Schlüsselworte ZZZ und XYZ äquivalent für dasselbe Argument verwendet werden.

Argumente können notwendig oder freiwillig (optional) sein und können auf zwei Arten angegeben werden:

Durch ihre Lage: die Argumente müssen genau in der angegebenen Reihenfolge stehen.

Durch Schlüsselwörter: die Reihenfolge spielt keine Rolle, da jedem Argument ein Schlüsselwort vorausgeht.

Wenn zum Beispiel der Befehl KOMMANDO aus einer Datei liest und in eine andere schreibt, heißen die Argumente

```
FROM, TO
```

Der Befehl kann nur durch die Position der Dateien erteilt werden (Schlüsselwörter werden nicht benötigt):

```
KOMMANDO eingabedatei ausgabedatei
```

Sie können aber auch Schlüsselwörter verwenden:

```
KOMMANDO FROM eingabedatei TO ausgabedatei
```

Mit den Schlüsselwörtern können Sie auch die Reihenfolge der Argumente umkehren:

```
KOMMANDO TO ausgabedatei FROM eingabedatei
```

Sie können aber auch beide Formen miteinander kombinieren:

```
KOMMANDO eingabedatei TO ausgabedatei
```

Dabei wird das FROM-Argument durch seine Position und das TO-Argument durch sein Schlüsselwort bestimmt. Folgende Form ist jedoch nicht zulässig:

```
KOMMANDO ausgabedatei FROM eingabedatei
```

weil der Befehl voraussetzt, daß AUSGABEDATEI das erste Argument ist und durch die Lage bestimmt wird (ein Schlüsselwort ist ja nicht vorhanden), also eine FROM-Datei darstellt.

Wenn das Argument nicht nur ein einzelnes Wort ist, sondern aus zwei Teilen besteht, die durch Leerzeichen getrennt werden, muß das ganze Argument zwischen zwei Anführungszeichen (") stehen. Wenn das Argument gleich einem Schlüsselwort ist, muß es zur Unterscheidung ebenfalls in Anführungszeichen stehen. Im folgenden Beispiel bezeichnet die Textdatei »dateiname« als das FROM-Argument und den Dateinamen »from« als das TO-Argument:

```
KOMMANDO "dateiname" TO "from"
```

Schlüsselwörter in den Argumenten-Listen haben Kennzeichner, die mit den Schlüsselwörtern verbunden sind. Diese Kennzeichner werden durch einen Schrägstrich und einen speziellen Buchstaben angezeigt. Die Bedeutungen der Kennzeichner sind wie folgt:

/A Das Argument wird benötigt und darf nicht weggelassen werden

/K Das Argument muß mit einem Schlüsselwort eingegeben und darf nicht durch seine Position gekennzeichnet werden.

/S Das Schlüsselwort verwendet keine Argumente, da es als VERMITTLER dient.

Die Kennzeichner /A und /K können kombiniert werden. Die folgende Beschreibung bedeutet zum Beispiel, daß sowohl das Schlüsselwort DRIVE als auch das Argument angegeben werden muß.

```
DRIVE/A/K
```

Betrachten Sie schließlich auch das Format des Befehls TYPE:

FROM/A, TO, OPT/K

Das erste Argument kann also nach Position oder mit Schlüsselwort angegegben werden. Das zweite muß mit Schlüsselwort eingeleitet werden. Argumente, die nach OPT folgen, sind optional, das heißt, sie können bei Bedarf verwendet werden. Wird OPT jedoch verwendet, muß auch das Schlüsselwort angegeben werden. Zur Verdeutlichung einige Beispiele:

```
TYPE Dateiname
TYPE FROM Dateiname
TYPE Dateiname TO Ausgabedatei
TYPE Dateiname Ausgabedatei
```

```
TYPE TO Ausgabedatei FROM Dateiname OPT n
TYPE Dateiname OPT n
TYPE Dateiname OPT n TO Ausgabedatei
```

Dieses Handbuch zeigt bei jedem Befehl alle möglichen Schlüsselworte und Argumente auf. Sollten Sie dennoch einmal an der korrekten Syntax zweifeln, tippen Sie nur den Befehl ein, dahinter ein Leerzeichen und dann ein Fragezeichen. AmigaDOS zeigt Ihnen dann die korrekte Form. Sie müssen nicht immer im Handbuch nachlesen. Haben Sie nicht die richtige Form gefunden, bricht AmigaDOS die Ausführung des Befehls mit der simplen Meldung BAD ARGS oder BAD ARGUMENTS ab. Der gesamte Befehl muß dann neu eingetippt werden.

1.5 Der Aktualisierungsprozeß

Haben Sie eine Diskette eingelegt, erzeugt AmigaDOS dafür einen Prozeß mit niedriger Priorität. Dieser überprüft den ganzen Aufbau der Diskette. Bis der Aktualisierungsprozeß zu Ende ist, können keine Dateien auf dieser Diskette erzeugt werden. Ein Lesen vorhandener Dateien ist jedoch möglich.

Ist der Aktualisierungsprozeß erledigt, kontrolliert AmigaDOS, ob das Datum und die aktuelle Uhrzeit für das System gesetzt wurden. Datum und Uhrzeit werden mit dem Befehl DATE eingestellt. Setzen Sie kein Datum oder keine Uhrzeit für das System, so nimmt AmigaDOS dafür die Daten der zuletzt erstellten Datei auf der eingelegten Diskette. Dies gibt die Gewißheit, daß neuere Dateiversionen als solche zu erkennen sind, auch wenn das aktuelle Datum und die Zeitangabe nicht stimmen sollten.

1.6 Die gebräuchlichsten Befehle: Eine kleine Übersicht

Dieses Handbuch beschreibt die verschiedenartigsten Anweisungen von AmigaDOS. Das CLI liest die eingetippten Anweisungen und übersetzt sie in vom Computer verstandene Symbole. In diesem Sinne entspricht AmigaDOS mehr einer herkömmlichen Schnittstelle als die Workbench.

Um die nun folgenden Anweisungen auch ausprobieren zu können, müssen Sie auf der Workbench das Programm PREFERENCES starten und dort, wie in der Einführung beschrieben, das CLI aktivieren.

1.6.1 Einführung in einige AmigaDOS-Anweisungen

Obwohl alle Anweisungen von AmigaDOS in Kapitel 2 dieses Buches in allen Einzelheiten erklärt werden, haben wir eine Zusammenstellung aller wichtigen Anweisungen in ihrer

meist verwendeten Form erstellt. Denn die meisten Anwender werden die vielen zusätzlichen Argumente, die zur vollständigen Syntax eines Befehls gehören, niemals benutzen.

Die unten aufgeführten Anweisungen (zusammen mit ihrem Schlüsselwort) bewirken in AmigaDOS so wichtige Dinge wie:

- eine Diskette kopieren (DISKKOPY)
- eine Diskette formatieren (FORMAT)
- eine Diskette startfähig machen (INSTALL)
- eine Diskette umbenennen (RELABEL)
- das Directory einer Diskette lesen (DIR)
- Informationen über Files lesen (LIST)
- ein File schützen (PROTECT)
- Informationen über eine Diskette lesen (INFO)
- das aktuelle Directory ändern (CD)
- Datum und Zeit neu setzen (DATE)
- das Ergebnis eines Befehls ***umleiten*** (>)
- ein Textfile auf dem Bildschirm ausgeben (TYPE)
- ein File umbenennen (RENAME)
- ein File löschen (DELETE)
- Files kopieren mit zwei Laufwerken (COPY)
- Files kopieren mit einem Laufwerk (COPY)
- ein neues Directory einrichten (MAKEDIR)
- Files auf einer Diskette finden (DIR OPT A)
- Anweisungen beim Systemstart automatisch durchführen (Startup-Sequenzen)
- logische Devices erzeugen, ändern oder löschen (ASSIGN)
- ein neues CLI-Fenster öffnen (NEWCLI)
- ein CLI-Fenster schließen (ENDCLI)

1.6.2 Tip für CLI-Neulinge

Wir empfehlen Ihnen, diesen Teil des Buches Schritt für Schritt durchzuarbeiten und jedes einzelne Beispiel nachzuvollziehen. Alle Beispiele sind auf Lauffähigkeit überprüft und leicht überschaubar. Wenn Sie mit dem System dann mehr vertraut sind, können Ihnen die Beispiele als Gedächtnisstütze dienen.

1.6.3 Zu Beginn

Bevor Sie beginnen, sollten Sie Ihre Workbench- oder CLI-Diskette und zwei neue, unbeschriebene Disketten bereitlegen. Bitte bringen Sie nun den Schreibschutz an Ihrer Systemdiskette an!

Die meisten unten beschriebenen Anweisungen gelten sowohl für Systeme mit einem als auch für Systeme mit zwei Laufwerken. Unterscheidet sich die Syntax, wurden beide Versionen abgedruckt.

Nach Eingabe des jeweiligen Befehls drücken Sie bitte die Return-Taste. Damit übernimmt der Computer die Ausführung des Befehls. Alle Schlüsselworte und Argumente sind in Großbuchstaben gedruckt. Dies dient nur der besseren Überschaubarkeit der Beispiele. AmigaDOS erkennt sie auch, wenn die gesamte Zeile mit kleinen Buchstaben getippt wird.

Der Name DFO: oder *erstes Laufwerk* bezieht sich in unseren Beispielen auf das eingebaute Diskettenlaufwerk des Amiga 1000 und des Amiga 500 oder das rechte Laufwerk des Amiga 2000, mit DF1: oder *zweites Laufwerk* ist das erste externe 3 1/2"-Laufwerk des Amiga 1000 und des Amiga 500 oder das rechte Laufwerk des Amiga 2000 bezeichnet. Ein Semicolon (;) in einer Anweisungszeile grenzt den Kommentar dahinter von dem eigentlichen Befehl ab. Alles, was hinter dem (;) steht, wird von AmigaDOS ignoriert. Diese Kommentare dienen nur dem Verständnis und brauchen nicht abgetippt zu werden.

1.6.4 Eine Diskette kopieren

Mit diesem Befehl werden ganze Disketten kopiert.

Mit einem Diskettenlaufwerk:

```
DISKCOPY FROM df0: TO df0:
```

mit zwei Laufwerken:

```
DISKCOPY FROM df0: TO df1:
```

Folgen Sie nun den auf dem Bildschirm erscheinenden Anweisungen:

Kopieren mit einem Laufwerk: Zuerst schieben Sie die schreibgeschützte Ursprungsdiskette (FROM- oder SOURCE-Disk) in das Laufwerk, dann drücken Sie <Return>. Der Amiga liest nun einen Teil des Disketteninhalts in seinen Speicher. Dann fordert Sie der Rechner auf, die Zieldisk (TO- oder DESTINATION-Disk) einzulegen und wieder <Return> zu drücken. Nun schreibt der Rechner den Inhalt seines Speichers auf die neue Diskette. Diese beiden Vorgänge wiederholen sich nun einige Male, bis der gesamte Disketteninhalt übertragen ist.

Kopieren mit zwei Laufwerken: Schieben Sie die schreibgeschützte Ursprungsdiskette (FROM- oder SOURCE-Disk) in das interne Laufwerk und die Zieldisk (TO- oder DESTINATION-Disk) in das externe Laufwerk. Dann drücken Sie die <Return>-Taste. Alles Weitere können Sie getrost dem Computer überlassen.

Kopieren Sie nun bitte Ihre CLI-Diskette und verwahren Sie das Original sicher. Schieben Sie nun die Kopie der Startdiskette in das interne Laufwerk und starten Sie das System von

neuem. (Wie das genau geht, können Sie im Handbuch Ihres Amiga nachlesen.) Starten Sie nun das CLI!

1.6.5 Eine Diskette formatieren

Vergewissern Sie sich bitte, daß Ihre CLI-Diskette in Laufwerk DFO: eingelegt ist und Sie eine neue Diskette zur Verfügung haben.

Mit dem Befehl FORMAT bereiten Sie eine neue Diskette zur Aufnahme von Daten oder Programmen vor. Auf diese Diskette können später einzelne Files kopiert werden. Tippen Sie nun:

```
FORMAT DRIVE df0: NAME Ein.Name
```

Folgen Sie nun wieder den Anweisungen auf dem Bildschirm. Legen Sie die neue, leere Diskette in das interne Laufwerk ein und drücken Sie dann Return. Der Amiga kann in allen angeschlossenen Laufwerken Disketten formatieren. Dazu wird nur der jeweilige Device-Name eingesetzt (DFO:-DF3:). Ist der Formatiervorgang abgeschlossen, geht die Lampe am Laufwerk aus. Entnehmen Sie nun bitte die neu formatierte Diskette. Legen Sie nun wieder Ihre CLI-Diskette in das Laufwerk DF0:.

1.6.6 Eine Diskette startfähig machen

Für diesen Versuch sollten Sie die Kopie der CLI-Diskette und die neu formatierte Diskette bereitlegen.

Es gibt mehrere Wege, eine startfähige Diskette zu erzeugen, zwei davon stellen wir Ihnen hier vor. Als Startdiskette bezeichnet man eine Diskette, die eingelegt werden kann, wenn der Amiga eine Workbench-Diskette verlangt. Eine formatierte Diskette wird zur CLI-Diskette, wenn Sie tippen:

```
INSTALL ?
```

Wenn Sie nur ein Laufwerk benutzen, müssen Sie das Fragezeichen eintippen. AmigaDOS versucht sonst sofort, die Diskette in df0: zu installieren, und das ist Ihre Startdiskette!

AmigaDOS antwortet nun:

```
DRIVE/A:
```

entnehmen Sie nun Ihre CLI-Diskette und legen Sie die neu formatierte ein. Dann tippen Sie:

```
df0:
```

AmigaDOS kopiert nun die sogenannten *Boot-Sektoren* auf die Diskette. Ist die Lampe am Laufwerk erloschen, führen Sie bitte einen *full reset* (Ctrl-*A*-*A*) durch. Nach dem Startvorgang haben Sie nun direkt ein CLI-Task zur Verfügung. Aber nur der Interpreter ist

auf der Diskette, der allein kann keine der hier beschriebenen Anweisungen ausführen, denn die Erklärungen und Abläufe der Anweisungen stehen im Directory C auf der Workbench-Diskette. Das müßten Sie nun aber erst auf diese Diskette kopieren.

Die zweite Methode, eine startfähige Diskette zu produzieren, ist weit sinnvoller, denn sie erhält das *Command-Directory*. Hier nun eine Schritt-für-Schritt-Anleitung für die Umwandlung einer Workbench-Diskette in eine CLI-Diskette:

1. Kopieren Sie Ihre Workbench-Diskette
2. Öffnen Sie ein CLI wie bereits beschrieben
3. Clicken Sie in das CLI-Fenster und tippen Sie nun:

```
RENAME FROM s/startup-sequence TO s/NO-startup-sequence
```

Warten Sie nun bitte das Erlöschen der LED am Laufwerk ab, und führen Sie wieder einen »full reset« durch. Der Amiga zeigt nun sofort ein aktives CLI-Fenster. Um aus dieser Diskette wieder eine Workbench-Diskette zu machen, geben Sie den oben gezeigten Befehl mit vertauschten Argumenten ein. Nach dieser Umwandlung stehen Ihnen die deutschen Sonderzeichen erst zur Verfügung, wenn Sie mit SYSTEM/SETMAP D die Belegung intern geändert haben!

1.6.7 Eine Diskette umbenennen

Ihre CLI-Diskette sollte nun wieder in Laufwerk DF0: eingelegt sein. Jeder Diskette mit AmigaDOS-Format kann jederzeit ein neuer Name zugeteilt werden. Dazu dient der Befehl RELABEL. Tippen Sie nun:

```
RELABEL Ein.Name: Ein.neuer.Name
```

In diesem Beispiel haben wir der vorhin formatierten Diskette einen neuen Namen zugewiesen.

1.6.8 Das Directory einer Diskette lesen

Ihre CLI-Diskette sollte nun wieder in Laufwerk DF0: eingelegt sein. Mit diesem Befehl lesen Sie das Inhaltsverzeichnis der eingelegten Diskette:

```
DIR oder DIR df0:
```

Diese Form listet den Inhalt des aktuellen Directory. Den Inhalt eines anderen Directory können Sie lesen, wenn Sie den Pfadnamen des Directory eingeben. Für das Directory C also:

```
DIR df0:c oder DIR c
```

Anstelle der Laufwerksbezeichnung kann auch der Diskettenname verwendet werden. Der Inhalt der eben formatierten leeren Diskette wird also mit:

```
DIR Ein.neuer.Name:
```

auf den Bildschirm gebracht. Heben Sie sich das aber bis später auf, die Disk ist jetzt noch leer.

1.6.9 Benutzung des LIST-Befehls

Ihre CLI-Diskette sollte nun wieder in Laufwerk DF0: eingelegt sein.

Mit dem Befehl DIR werden die Namen der Files im aktuellen Directory ausgelesen. Der Befehl LIST macht nun zusätzliche Informationen zugänglich. Tippen Sie ein:

```
LIST oder LIST df0:
```

AmigaDOS gibt daraufhin Informationen über alle Files in das jeweilige Directory aus. So zum Beispiel Informationen über die Größe des Files, über dessen Status (löschbar oder nicht) und über das Datum und die Uhrzeit, zu der es erstellt wurde. Geben Sie zum Befehl LIST und dahinter den Namen eines Directory an, werden alle Files dieses Directory angezeigt.

```
LIST c
```

zeigt alle Files des *Anweisungsverzeichnisses* von AmigaDOS.

Mit den Buchstaben »wred« wird der Status eines Files gezeigt. Dabei steht »r« für lesen (read), »w« für schreiben (write), »e« für ausführbar (execute) und »d« für löschbar (delete). AmigaDOS achtet sehr genau auf diese *flags*. Erscheint zum Beispiel das flag »d« hinter einem File nicht, kann dieses mit dem Befehl DELETE nicht gelöscht werden.

1.6.10 Ein File schützen

Ihre CLI-Diskette sollte nun wieder im Laufwerk DF0: eingelegt sein.

Der Befehl PROTECT schützt ein File vor dem versehentlichen Löschen. Auch die Rücknahme dieses Schutzes übernimmt PROTECT. Tippen Sie:

```
DATE > Testfile
PROTECT Testfile
LIST Testfile
```

Nun sind alle Flags gesetzt als »–«. Wenn Sie nun versuchen, dieses File mit dem folgenden Befehl zu löschen:

```
DELETE Testfile
```

antwortet AmigaDOS:

```
Not Deleted - file is protected from deletion
```

Sinngemäß bedeutet das etwa: *Nicht gelöscht - File ist vor dem Löschen geschützt.*

Um nun unser File wieder löschen zu können, muß der Schutz mit der folgende Anweisung entfernt werden:

```
PROTECT Testfile d oder PROTECT Testfile rwed
```

1.6.11 Informationen über eine Diskette lesen

Ihre CLI-Diskette sollte nun wieder im Laufwerk DF0: eingelegt sein.

Tippen Sie nun folgenden Befehl ein:

```
INFO
```

Auf dem Bildschirm erscheint alles Wissenswerte über die CLI-Diskette. Neben dem Diskettennamen und dem Status der verschiedenen Flags wird auch der freie und schon belegte Speicherplatz angezeigt. Benutzen Sie nur ein Laufwerk, erfahren Sie mehr über eine momentan nicht eingelegte Diskette, wenn Sie

```
INFO ?
```

tippen. AmigaDOS antwortet

```
none:
```

AmigaDOS hat nämlich auf Ihren Befehl hin den Befehl von der CLI-Diskette geladen und zeigt die benötigten Argumente – und das sind keine (auf englisch *none*).

Entfernen Sie nun die CLI-Diskette aus dem Laufwerk und legen Sie die gewünschte ein. Drücken Sie nun noch die Return-Taste, zeigt AmigaDOS alle Informationen über die eben eingelegte Diskette.

1.6.12 Das aktuelle Directory ändern

Bis jetzt haben wir uns nur in der obersten Ebene des hierarchischen Filesystems von AmigaDOS bewegt. Mehr Informationen über Directory-Strukturen finden Sie in Kapitel 1.3 dieses Handbuches. Mit dem Befehl:

```
CD
```

erfahren Sie, welches momentan Ihr aktuelles Directory ist. Haben Sie die CLI-Diskette im internen Laufwerk und geben dann den Befehl DIR ein, sehen Sie unter anderem auch den Eintrag:

```
c (dir)
```

Dieses Directory soll nun das aktuelle werden. Geben Sie dazu ein:

```
CD C oder CD df0:c
```

Lesen Sie nun nochmals das Directory, dann werden Sie feststellen, daß nur der Inhalt des Directory C aufgelistet wird.

In die oberste Fileebene einer Diskette kommen Sie wieder, wenn Sie dem Befehl einen Doppelpunkt folgen lassen:

```
CD : oder CD df0:
```

1.6.13 Datum und Zeit neu setzen

Die AmigaDOS-Uhr wird mit dem Befehl DATE gestellt:

```
DATE 12:00:00 12-jan-87
```

Die Systemuhr enthält jetzt diese Zeit.

1.6.14 Das Ergebnis eines Befehls umleiten

Normalerweise wird jedes von einem Befehl gelieferte Ergebnis auf dem Monitor angezeigt. Sie können diese Ausgabe jedoch mit dem Zeichen »>« statt dessen in ein File schreiben. Hier ein Beispiel:

```
DATE > Datenfile
```

Hat der Rechner diesen Befehl ausgeführt, finden Sie im aktuellen Directory einen Eintrag »Datenfile«. Existierte dieses File bereits vorher, wurde der alte Inhalt überschrieben! Mit der Syntax:

```
DATE > Ein.neuer.Name:Datenfile
```

wird das File auf die vorhin formatierte Diskette geschrieben. Dazu werden Sie von AmigaDOS aufgefordert, diese Diskette in das Laufwerk einzulegen. Warten Sie dann das Erlöschen der LED am Laufwerk ab und legen Sie die CLI-Diskette wieder ein. Tippen Sie nun:

```
DIR Ein.neuer.Name:
```

Wieder werden Sie zum Diskettenwechsel aufgefordert. Nun finden Sie auf der neuen Diskette ebenfalls ein File mit Namen DATENFILE.

1.6.15 Ein Textfile auf dem Bildschirm ausgeben

Der Inhalt eines Textfiles kann mit dem Befehl TYPE lesbar gemacht werden:

```
TYPE Datenfile
```

Wollen Sie die Ausgabe vorübergehend anhalten, drücken Sie einfach auf die Leertaste. Weiter geht sie dann mit <Backspace>. Soll die Ausgabe vor dem Fileende abgebrochen werden, halten Sie bitte die CTRL-Taste gedrückt und tippen ein »c«.

Um ein Testfile auf einer nicht aktuellen Diskette zu lesen, tippen Sie:

```
TYPE Ein.neuer.Name:Datenfile
```

1.6.16 Ein File umbenennen

Der Name eines Files kann mit dem Befehl

```
RENAME FROM Datenfile TO neuer.Name
```

oder einfach mit

```
RENAME Datenfile neuer.Name
```

geändert werden.

Beachten Sie bitte, daß die meisten AmigaDOS-Anweisungen in dieser abgekürzten Form verwendet werden dürfen. Zu Beginn Ihrer Arbeit mit AmigaDOS empfehlen wir aber die lange Variante mit Schlüsselwörtern, da deren Funktion leichter zu durchschauen ist. In der Kurzübersicht sind alle alternativen Formen aufgezeigt.

Überprüfen Sie nun mit TYPE, daß sich der Inhalt des Files nicht geändert hat:

```
TYPE neuer.Name
```

1.6.17 Ein File löschen

Um wieder Platz auf Ihren AmigaDOS-Disketten zu schaffen, werden Sie von Zeit zu Zeit nicht mehr genutzte Files oder Programme löschen wollen. Dies ermöglicht der Befehl DELETE.

> **Achtung!** Ein einmal gelöschtes File kann nicht wieder herbeigeholt werden! Vergewissern Sie sich, ob das File wirklich nutzlos ist.

Hier nun eine kleine Beispielsequenz. Sie erstellt ein kleines Textfile, gibt dieses File auf dem Bildschirm aus, löscht es und versucht es dann erneut zu lesen:

```
DIR    > directoryfile
TYPE   directoryfile
DELETE directoryfile
TYPE   directoryfile
```

Nach dem letzten Befehl sehen Sie auf dem Bildschirm:

```
Can't open directoryfile
```

Das heißt auf deutsch etwa: *ich kann directoryfile nicht öffnen.*

Fragen Sie danach *warum* (englisch *why*):

```
WHY
```

Dann antwortet AmigaDOS darauf mit:

```
Last command failt because object not found
```

Das heißt auf deutsch etwa: *Der letzte Befehl kann nicht ausgeführt werden, weil es das Objekt nicht gibt.*

1.6.18 Files kopieren mit zwei Laufwerken

Mit zwei Laufwerken ist das Kopieren von Files ein Kinderspiel:

```
COPY FROM df0:ursprungfile TO df1:zielfile
```

oder

```
COPY df0:ursprungfile df1:zielfile
```

Selbstverständlich können Sie auch ein File von df1: nach Laufwerk df0: kopieren.

1.6.19 Files kopieren mit einem Laufwerk

Mit einem Laufwerk Files zu kopieren erfordert etwas mehr Arbeit. Zuerst muß eine *RAM-Disk* erstellt und ein Teil des Directory C auf diese kopiert werden. Eine RAM-Disk stellt AmigaDOS als RAM:-Device zur Verfügung. Und so wird es gemacht:

```
COPY df0:c/cd    RAM:
COPY df0:c/copy RAM:
CD RAM:
```

Legen Sie dann die Ursprungsdiskette in das Laufwerk. (Für dieses Beispiel kopieren wir das File EXECUTE aus dem Directory C von der CLI-Diskette, die bereits im Laufwerk ist.) Tippen Sie:

```
COPY df0:c/execute RAM:execute
```

oder

```
COPY df0:c/execute execute
```

oder

```
COPY df0:c/execute RAM:
```

Entnehmen Sie nun die Ursprungsdiskette dem Laufwerk und legen Sie die Zieldiskette ein. Tippen Sie:

```
COPY RAM:execute df0:execute
```

oder

```
COPY execute df0:execute
```

Entnehmen Sie nun die Zieldiskette wieder und schieben Sie Ihre CLI-Diskette in das Laufwerk. Tippen Sie nun:

```
CD df0:
```

und Sie sind wieder am Anfang.

Mit dem folgenden Befehl wird auch der Speicherplatz, den die RAM-Disk belegte, wieder frei:

```
DELETE RAM:cd RAM:copy RAM:execute
```

(Allerdings bleibt dabei 1 Kbyte für das Directory der RAM-Disk zurück. Ein Verlust, der zu verschmerzen ist.)

1.6.20 Ein neues Directory einrichten

Ein neues Directory (eine neue Schublade) im aktuellen Directory wird mit dem Befehl

```
MAKEDIR neudirectory
```

eingerichtet.

Um ein File aus einem Directory in ein anderes auf der gleichen Diskette zu befördern, benutzen Sie den Befehl

```
RENAME FROM dateiname TO neudirectory/dateiname
```

Mit dem Befehl

```
DIR neudirectory
```

können Sie sich von der Funktion überzeugen.

1.6.21 Files auf einer Diskette finden

Manchmal möchte man nicht nur den Inhalt eines Directory, sondern den Inhalt der ganzen Diskette durchsehen. Diesem Zweck dient der Befehl DIR mit einer deren zusätzlichen Optionen. Geben Sie den folgenden Befehl ein:

```
DIR OPT A
```

und alle Directories mit allen Unterdirectories und all deren Files erscheinen am Bildschirm. (Mit der Leertaste wird die Ausgabe angehalten, mit <Backspace> fortgesetzt!)

Um sich einen genaueren Überblick zu verschaffen, leitet man die Ausgabe in ein File um:

```
DIR > directoryfile OPT A
```

Wichtig ist, daß die Optionen nach dem Filenamen eingegeben werden.

Nun können Sie das File mit *Type* schnell auf den Bildschirm holen oder es mit dem Bildschirm-Editor lesen und bearbeiten. Machen Sie mit:

```
ED directoryfile
```

Mit den Cursor-Tasten können Sie nun vor- und zurück-»blättern«.

Mit der Tastenkombination ESC T und <Return> geht's an den Anfang. Diese Tastenkombination wird auch ESC-T genannt.

Mit der Tastenkombination ESC B <Return> geht's zum Ende der Datei.

Mit der Tastenkombination ESC M, einer Zahl »n« und <Return> geht's zur Zeile mit der Nummer »n«.

Mit der Tastenkombination ESC Q <Return> wird die Arbeit beendet, ohne das File auf Diskette zurückzuschreiben. Haben Sie am File etwas verändert, verlangt AmigaDOS die Eingabe von Y, bevor es den Editor verläßt. Und mit der Tastenkombination ESC X <Return> werden die Änderungen gespeichert.

In Kapitel 3 dieses Handbuches wird ED aber noch genauer beschrieben.

1.6.22 Anweisungen beim Systemstart automatisch durchführen lassen

Im Directory S der CLI-Diskette befindet sich ein File namens STARTUP-SEQUENCE.

Dies ist ein sogenanntes *Execute-File*, also eine Ansammlung von CLI-Anweisungen, die automatisch ausgeführt werden, wenn das System gestartet wird. Die beiden letzten Anweisungen dieses Files sind LOADWB (lädt Workbench) und ENDCLI (beendet den CLI-Task und übergibt die Kontrolle des Systems an die Workbench). Mit den Programmen ED und EDIT können Sie nun eine eigene Startup-Sequence erzeugen. Später wird genauer auf den Befehl EXECUTE eingegangen. Durch eine geänderte Startup-Sequence können Programme sofort beim Systemstart geladen und gestartet werden.

> **Achtung!** Die sicherste Methode, Ihre Original-Workbench-Diskette zu ruinieren, ist die, Versuche mit einer geänderten Startup-Sequence mit ihr zu machen! Nehmen Sie eine Kopie.

1.6.23 Logische Devices erzeugen, ändern oder löschen

Manchmal möchten Sie mit einer anderen als der Startdiskette weiterarbeiten. Haben Sie zum Beispiel den Rechner mit der Workbench-Diskette gestartet und wollen mit der Diskette EIN NEUER.NAME weiterarbeiten, geht das mit einem Laufwerk nur mit vertretbarem Aufwand, wenn auf der Diskette EIN.NEUER.NAME das C-Directory mit den Files für die Befehle des CLI zu finden ist. Um dann auf dieses File zugreifen zu können, verwenden Sie den Befehl ASSIGN. Verwenden Sie diesen Befehl nicht, sieht die weitere Arbeit so aus:

```
CD Ein.neuer.Name:
```

AmigaDOS antwortet INSERT EIN.NEUER.NAME INTO ANY DRIVE. Legen Sie also die Diskette EIN.NEUER.NAME in das Laufwerk, und tippen Sie dann:

```
DIR
```

AmigaDOS antwortet nun: INSERT CLIDISK (ODER IHRE STARTDISKETTE) INTO ANY DRIVE. Da AmigaDOS seit dem Start weiß, daß alle CLI-Anweisungen im Directory C auf der Startdiskette stehen, will es jetzt diese, und nur diese, Diskette haben. AmigaDOS liest also dann den Befehl ein und fordert Sie auf, die aktuelle Diskette einzulegen, also EIN.NEUER.NAME. Bei jedem weiteren CLI-Befehl sind nun ebenfalls zwei Diskettenwechsel nötig. Um dies zu vermeiden, tippen Sie folgendes:

```
ASSIGN c: Ein.neuer.Name:c
```

AmigaDOS fordert: INSERT EIN.NEUER.NAME INTO ANY DRIVE. Von nun an liest AmigaDOS seine Anweisungen aus dem Directory C der Diskette EIN.NEUER.NAME. Nun brauchen Sie nur noch den folgenden Befehl einzutippen und Sie sind am Ziel:

```
CD Ein.neuer.Name:
```

Es gibt noch mehr sinnvolle Zuweisungen unter AmigaDOS. Wenn Sie folgenden Befehl eingeben, werden diese aufgelistet:

```
ASSIGN LIST
```

Sprechen Sie zum Beispiel aus einem Programm heraus ein am seriellen oder parallelen Port angeschlossenes Gerät (Drucker oder Modem) an, sucht AmigaDOS im Directory DEVS: nach den zur Ansprache dieser Geräte nötigen Anweisungen. Haben Sie alle Systemdirectories auf die neue Diskette kopiert, brauchen Sie die Startdiskette nicht mehr. Der Inhalt eines Execute-Files für die Diskette EIN.NEUER.NAME sieht dann so aus:

```
ASSIGN SYS:          Ein.neuer.Name:
ASSIGN S:            Ein.neuer.Name:s
ASSIGN DEYS:         Ein.neuer.Name:deys
ASSIGN L:            Ein.neuer.Name:l
ASSIGN FONTS:        Ein.neuer.Name:fonds
ASSIGN LIBS:         Ein.neuer.Name:libs
```

Um dieses Execute-File zu erstellen, tippen Sie:

```
COPY FROM * TO umschaltfile
```

Dann geben Sie jede einzelne Zeile ein. Nach der letzten Zeile drücken Sie schließlich CTRL-/. Das Sternchen (*) steht für das aktuelle Fenster und die Tastatur. Auf diese Weise können kleinere Files ohne ED oder EDIT erstellt werden.

1.6.24 Ein neues CLI-Fenster öffnen

AmigaDOS ist ein multitaskingfähiges System. Sie können also zu gleicher Zeit in mehreren Fenstern verschiedene Aufgaben erledigen lassen. Jede Aufgabe braucht jedoch ein eigenes Fenster. Ein neues CLI erscheint mit dem Befehl:

```
NEWCLI
```

Das *Prompt*, das in diesem Fenster erscheint, enthält vor dem »größer als-«Zeichen (>) die laufende Nummer des CLI-Fensters:

```
1>
```

Geben Sie nun ein:

```
NEWCLI
```

Daraufhin erscheint ein neues Fenster mit dem folgenden Prompt

```
2>
```

Dieses Fenster kann nun größer oder kleiner gemacht und über den ganzen Bildschirm bewegt werden.Wollen Sie dem zweiten Task Anweisungen erteilen, klicken Sie in das Fenster und tippen den Befehl ein. Versuchen Sie folgendes:

1. Klicken Sie in Fenster *1>* und tippen Sie:

```
DIR df0:c
```

2. Klicken Sie schnell in Fenster *2>* und dann:

```
INFO
```

Beide CLI-Fenster erfüllen ihre Aufgaben zur selben Zeit. Sie sind dabei nicht auf diese zwei Fenster beschränkt. Steht genug Speicherplatz zur Verfügung, können 20 oder mehr Fenster gleichzeitig geöffnet sein.

1.6.25 Ein CLI-Fenster schließen

Der jeweilige CLI-Task wird beendet, indem Sie mit dem Auswahl-Knopf der Maus in das Fenster Klicken und dann den Befehl

```
ENDCLI
```

eingeben. Das war's schon; das Fenster verschwindet daraufhin vom Bildschirm.

1.6.26 Zum Schluß dieses Abschnittes

Diese Beispiele sollten Ihnen nur einen kleinen Überblick über die Fähigkeiten von AmigaDOS und CLI geben. Sehr viele der Anweisungen haben wir in diesem Teil des Buches nicht angeschnitten, bei den anderen haben wir die Syntax stark vereinfacht und viele Optionen weggelassen.

Kapitel 2 enthält die genaue Beschreibung und alle Optionen zu jedem einzelnen Befehl. Sehen Sie bitte dort nach, wenn etwas an den Beispielen unklar geblieben ist. Als geübter Anwender mag Ihnen die Kurzübersicht am Ende von Kapitel 2 als Gedächtnisstütze dienen.

1.7 Vereinbarungen zum Befehlsteil

Im Kapitel 2 dieses Handbuches wird für die Beschreibung der einzelnen Anweisungen folgende Syntax verwandt:

Ein *Platzhalter* wie <name> verlangt einen Parameter mit der Bezeichnung *name*.

Zum Beispiel: *EXECUTE <befehlsfile>* benötigt den Namen des von Ihnen erstellten Befehlsfiles.

Eckige Klammern (»[« und »]«) bezeichnen optionale Argumente. Dieses Argument wird für die Funktion des Befehls nicht benötigt. Soll es aber angegeben werden, ist die korrekte Syntax erforderlich. Ein Beispiel für die Verwendung eckiger Klammern ist QUIT [<code>].

Ein senkrechter Strich »|« grenzt alternativ verwendbare Argumente ab. Eines der möglichen Argumente muß angegeben werden. Lautet die Beschreibung DIR [OPT A|I|AI], so können Sie zwischen den Argumenten OPT A, OPT I und OPT AI wählen.

Ein Sternchen hinter einem Platzhalter (wie bei <name>*) zeigt an, daß einer oder mehrere Namen eingegeben werden können. Jeder Name muß mit einem Leerzeichen von anderen Namen abgetrennt werden.

Jeder einzelne Parameter eines AmigaDOS-Befehls muß durch ein Leerzeichen von dem nächsten getrennt werden. Lassen Sie sich nicht von den scheinbaren Unterschieden zwischen *Syntax* und *Muster* verwirren. In der Rubrik *Muster* sind aus praktischen Gründen nicht überall Leerzeichen eingefügt. Näheres über das *Muster* von Anweisungen finden Sie in Abschnitt 1.4.5 dieses Buches.

Kapitel 2: Die AmigaDOS-Befehle

Dieses Kapitel ist in zwei Abschnitte unterteilt: Der erste Abschnitt beschreibt die Anwenderbefehle, die der Amiga zur Verfügung stellt; der zweite die Befehle für Programmentwickler. Die Anwenderbefehle sind in verschiedene Kategorien unterteilt: Dateihilfen, Gebrauch des CLI, Gebrauch von Befehlssequenzen sowie System- und Speicherverwaltung. In den Abschnitten 1 und 2 werden die Syntax und der Zweck eines jeden Befehls beschrieben. Zusätzlich werden Beispiele und Querverweise gegeben.

Das Kapitel beginnt mit einer Zusammenstellung der verwendeten Fachwörter. Am Ende des Kapitels findet sich eine Kurzübersicht zum Nachschlagen der Befehle und ihrer Funktionen.

2.1 Anwender-Befehle des AmigaDOS

2.2 AmigaDOS-Befehle für Programmierer

2.3 Alphabetische Kurzübersicht über die Befehle von AmigaDOS

Verwendete Fachausdrücke

In diesem Handbuch finden sich einige Begriffe, die manchem Leser unbekannt sein könnten. Darum eine kleine Erklärung der wichtigsten Fachausdrücke:

Booten/Bootvorgang Starten des Rechners und Einlesen des Betriebssystems.

Default Eine Grundeinstellung beziehungsweise die Einstellung, die gültig ist, wenn Sie nichts anderes bestimmen.

Device Ein *Device* ist ein Gerät, wie ein Floppylaufwerk oder der Drucker. Damit kann man aber auch eine rechnerinterne Adresse bezeichnen, die wie ein physikalisches Gerät agiert, wie zum Beispiel eine RAM-Disk.

Device-Name	Der Name, der ein Device von einem anderen unterscheidet und es bezeichnet. Device-Namen folgt immer ein Doppelpunkt (:). Beispiele: CON:, DF0:, PRT: und so weiter.
File handle	Ein interner Wert des DOS des Amiga, der eine offene Datei oder ein offenes Device repräsentiert.
Logisches Device	Ein Name, der einem Directory mit dem Befehl ASSIGN zugeordnet und dann wie ein Device verwendet werden kann.
Objekt Code	Binäre Ausgabe eines Assemblers oder Compilers (noch nicht lauffähiger Maschinencode) und binäre Eingabe an einen Linker.
Stream	Eine geöffnete Datei oder ein Device, die mit einem File-handle-Wert verbunden ist. Zum Beispiel kann der Eingabe-Stream aus einer Datei kommen und der Ausgabe-Stream zum Bildschirm (CON:) gehen.
System-Diskette	Die Diskette, auf der sich die Workbench und die CLI-Befehle befinden.
Volume-Name	Ein Name, der eine Diskette oder Festplatte bezeichnet.

2.1 Anwender-Befehle des AmigaDOS

Auf den folgenden Seiten finden Sie Beschreibungen der einzelnen Befehle, die Sie in einem CLI-Fenster verwenden können. Jede dieser Beschreibungen beginnt auf einer neuen Seite. Das Format der folgenden Befehlsbeschreibungen wird in den Abschnitten 1.4.5 und 1.7 genauer erklärt.

;

Format: [<anweisung>];[<kommentar>]

Muster: "anweisung";"kommentar"

Zweck: Einen Kommentar an einen Befehl anhängen.

Beschreibung: Das CLI ignoriert alles in einer Befehlszeile nach dem »;«.

Beispiel(e):

```
;Diese Zeile ist nur ein Kommentar
```

diese Zeile wird vom Rechner nicht beachtet.

```
COPY <file> TO prt: ;druckt das File
```

kopiert das File <file> zum Gerät Drucker und ignoriert den Teil: »;druckt das File«

Siehe auch: EXECUTE

><

Format: <anweisung> [>ausgabefilename] [<eingabefilename] [<anweisungsargumente>*]

Muster: "anweisung">"TO"<"FROM" "argumente"

Zweck: Umleiten der Ein- und Ausgaberichtung von Daten oder Argumenten.

Beschreibung: Die Symbole »<« oder »>« geben die Ausgabe- beziehungsweise Eingaberichtung eines Befehls an. Die Spitze des Zeichens weist auf die Richtung des Datenflusses hin. Die Ausgabe geht normalerweise in das aktuelle Fenster. Wird jedoch nach dem Befehl (also noch vor eventuellen Argumenten!) dieses Symbol »>« und ein Dateiname angegeben, werden die Ausgabedaten dieses Befehls in das File geschrieben. Existiert dieses File noch nicht, wird es neu eingerichtet (statt eines Files kann natürlich jedes andere Device Verwendung finden, zum Beispiel PRT:).

Erwartet der Befehl dagegen weitere Parameter und wird nach dem Befehl das Symbol »<« und ein Dateiname eingegeben, liest der Rechner alle weiteren Eingaben aus dem angegebenen File. Die Schlüsselworte TO und FROM brauchen nicht eingegeben zu werden. Die Änderung des Datenstromes gilt nur für diesen einen Befehl. Nach der Ausführung sind wieder Tastatur und aktuelles Fenster Ein- und Ausgabe-Devices.

Beispiel(e):

```
DATE > tagebuchdatei
```

Dieser Befehl schreibt das aktuelle Datum und die Uhrzeit in die Datei *tagebuchdatei*.

```
programm < input
```

Der Befehl *programm* übernimmt Daten von der Datei *input*.

```
LIST > temp
SORT temp TO *
```

Die Liste der Dateien wird in das File *temp* geschrieben und dann sortiert auf dem Bildschirm ausgegeben.

>< (Fortsetzung)

```
ECHO > 2.datum 02-jan-87
DATE < 2.datum ?
DELETE 2.datum
```

Erzeugt die Datei 2.DATUM mit dem Inhalt »02-jan-87 <linefeed>«, leitet dieses File des Befehls DATE als Argument zu und löscht dann das File wieder.

Bemerkung: Das Fragezeichen nach dem Dateinamen muß eingegeben werden, da DATE sonst nur Eingaben von der Tastatur annimmt.

ADDBUFFERS

Format: ADDBUFFERS DF<x>: <nn>

Zweck: Reduzierung der Disk-Zugriffszeiten durch Vergrößerung des Disk-Pufferspeichers.

Beschreibung: Dieser Befehl fügt <nn> Pufferspeicher zu der bestehenden Liste von Sektor-Zugriffspuffern für Drive <x> hinzu. Dadurch kann die effektive Disketten-Zugriffszeit teilweise erheblich reduziert werden. Der einzige Nachteil dieses Befehles: Er verbraucht Speicherplatz. Jeder neu hinzugefügte Arbeitspuffer benötigt ungefähr 500 Byte RAM.

Fügen Sie mehr als 25-30 zusätzliche Buffer hinzu, hat das kaum noch Auswirkungen auf die Zugriffszeiten.

Beispiel(e):

```
ADDBUFFERS DF1: 25
```

Hier werden 25 zusätzliche Puffer für Drive DF1: reserviert. Falls Sie nicht genau wissen, wieviel Pufferspeicher für Ihre Anwendung optimal ist, dann sollten Sie einmal bei diesem Wert starten.

ASSIGN

Format: ASSIGN [[<name>]<dir>] [LIST]

Muster: ASSIGN "NAME,DIR,LIST/S"

Zweck: Zuordnung logischer Devices.

Beschreibung: NAME ist der logische Device-Name, der dem Directory DIR zugewiesen wird. Geben Sie nur den Namen ein, wird das Device gelöscht. Der Befehl ASSIGN alleine oder mit dem Anhang LIST schreibt die Namen aller aktuellen Devices auf den Bildschirm. Beachten Sie bitte, daß die richtige Diskette in das Laufwerk eingelegt ist. ASSIGN arbeitet immer auf die Diskette bezogen, ohne zu berücksichtigen, in welchem Laufwerk sie eingelegt ist. ASSIGN bleibt nur bis zum REBOOT oder System-Neustart aktiv.

Beispiel(e):

```
ASSIGN ursprung: :neu/arbeit
```

Der logische Device-Name URSPRUNG wird dem Directory :NEU/ARBEIT zugewiesen.

Nun soll auf die Datei XYZ im Directory :NEU/ARBEIT auf dem Bildschirm ausgegeben werden. Dazu genügt der folgende Befehl:

```
TYPE ursprung: XYZ
```

```
ASSIGN LIST
```

Zeigt alle logischen Devices an. Dabei werden die vollen Directory-Namen sichtbar.

BINDDRIVERS

Format: BINDDRIVERS

Zweck: Einbinden von Devices für zusätzliche Hardware.

Beschreibung: Das Kommando BINDDRIVERS gehört eigentlich in eine Startup-Sequenz. Es wird dazu verwendet, Device-Driver, die sich im Directory mit dem Namen SYS:EXPANSION befinden und sich auf zusätzliche Hardware beziehen (zum Beispiel eine Festplatte), aufzurufen und damit automatisch einzubinden (für den Benutzer bedeutet das, daß Icons, die sich im Expansion-Directory der aktuellen Diskette befinden, automatisch konfiguriert werden).

Beispiel(e):

```
BINDDRIVERS
```

BREAK

Format: BREAK <task>[ALL] [C] [D] [E] [F]

Muster: BREAK "TASK/A, ALL/S, C/S, D/S, E/S, F/S"

Zweck: Setzen der Attention-Flags.

Beschreibung: BREAK setzt die einzeln anzugebenden Attention-Flags. C setzt das <Ctrl>-C-Flag, D das <Ctrl>-D-Flag und so weiter. ALL setzt alle angegebenen Flags. Wird kein Flag angegeben, setzt AmigaDOS nur das <Ctrl>-C-Flag. BREAK ist identisch mit dem Auswählen des Tasks mit der Maus, Anklicken des Fensters mit dem Auswahlknopf der Maus und der Eingabe der entsprechenden Ctrl-Tastenkombination. Der entsprechende Prozeß wird also meist unterbrochen.

Beispiel(e):

```
BREAK 7
```

setzt das <Ctrl>-C-Flag in Task 7

```
BREAK 5 D E
```

setzt die Flags <Ctrl>-D und <Ctrl>-E in Task 5.

CD

Format: CD[<dir>]

Muster: CD "DIR"

Zweck: Benennt das aktuelle Directory.

Beschreibung: CD dient dem Festsetzen oder Ändern des aktuellen Directory oder Laufwerks. Bei Eingabe von CD ohne Angabe von Parametern wird der Name des aktuellen Directory angezeigt (geben Sie doch einfach einmal CD gefolgt von einem <Return> ein). Folgt dem Befehl ein Directory-Name, bezeichnet er ein neues aktuelles Directory, in dem später Dateinamen ohne Angabe des Directory-Namens gesucht werden können. Befindet sich das genannte Directory nicht auf der Diskette im aktuellen Laufwerk, wechselt CD ebenfalls das aktuelle Laufwerk.

Um das jeweils übergeordnete Directory zum aktuellen zu machen (falls eines existiert), geben Sie nur CD gefolgt von einem einzelnen Schrägstrich (/) ein. Dieser Befehl »CD /« benennt das in der Hierarchie nächsthöhere Directory zum aktuellen, es sei denn, das aktuelle Directory ist schon das Quelldirectory, und es kann nicht mehr höher gestiegen werden. Mehrere Schrägstriche hintereinander sind erlaubt, für jeden Schrägstrich steigt das aktuelle Directory entsprechend eine Stufe höher.

Beispiel(e):

```
CD DF1:arbeit
```

macht das Directory ARBEIT auf der Diskette in Laufwerk 1 zum aktuellen. Das aktuelle Laufwerk wird dadurch ebenfalls DF1: zugewiesen.

```
CD SYS:COM/BASIC
CD /
```

ändert das aktuelle Directory zu SYS:COM.

CHANGETASKPRI

Format: CHANGETASKPRI <priorität>

Zweck: Ändern von CLI-Task-Prioritäten.

Beschreibung: Ihr Amiga verwendet sogenannte Prioritäten-Nummern zur Kennzeichnung derjenigen momentan stattfindenden Prozesse (Tasks), die er bedienen muß. Normalerweise laufen die meisten Benutzer-Tasks unter der Priorität 0. Die Rechenzeit der CPU wird dann zwischen den Prozessen gleicher Priorität »gerecht« geteilt.

Mit dem vorliegenden Befehl sind Sie nun in der Lage, die Priorität des aktuellen CLI-Tasks zu verändern. Alle Prozesse, die von diesem CLI-Task gestartet werden, behalten diese eingestellte Priorität. Sie können rein theoretisch Prioritäten-Nummern von -128 bis +127 angeben (es findet keine Werteüberprüfung statt). Damit bringen Sie allerdings einige Unordnung in die systeminternen Abläufe. Begnügen Sie sich deshalb möglichst mit Werten zwischen -5 bis 5.

Eine mögliche Anwendung wäre beispielsweise die gleichzeitige Arbeit mit einem Compiler und einem Editor. Hätten beide Prozesse (Compiler und Editor) gleiche Priorität, dann könnten Sie Ihren Text (oder Ihr Programm) nur sehr stockend editieren, da der Editor ständig vom Compiler unterbrochen wird. Geben Sie dem Editor jedoch eine höhere Priorität, dann werden Ihre Tastenbetätigungen bevorzugt bearbeitet. Der Compiler hat dann Zeit zwischen Ihren Tastenaktivitäten.

Beispiel(e):

```
CHANGETASKPRI 5
```

Der momentan aktuelle CLI-Prozeß und alle anderen, die von ihm aus gestartet werden, besitzen Priorität über alle anderen Benutzer-Tasks, die ohne Verwendung des Kommandos CHANGETASKPRI initialisiert wurden.

COPY

Format: COPY [[FROM]<name>] [TO<name>] [ALL] [QUIET]

Muster: COPY "FROM, TO/A, ALL/S, QUIET/S"

Zweck: Kopieren eines Files oder eines gesamten Directory.

Beschreibung: COPY kopiert eine Datei oder ein Directory von einem Platz zu einem anderen. COPY fertigt eine Kopie einer Datei oder eines Directory an und schreibt sie in eine mit TO genauer bezeichnete Datei oder in ein Directory. Der alte, vorher bestehende Inhalt der mit TO bezeichneten Datei wird, falls vorhanden, gelöscht.

Der Name hinter FROM muß angegeben werden. Das TO-Directory muß ebenfalls vorhanden sein, damit der Befehl COPY funktionieren kann, es wird von COPY nicht angelegt.

Hängen Sie den Zusatz ALL an, kopiert COPY alle Dateien aus allen Unterdirectories des FROM-Directory. In diesem Fall werden automatisch, soweit nötig, Unterdirectories in dem TO-Directory geschaffen. Der Name der aktuellen Datei, die kopiert wird, erscheint während des Vorgangs auf dem Bildschirm. Fügen Sie noch zusätzlich QUIET hinten an, so wird diese Ausgabe der aktuell kopierten Files unterdrückt.

Sie können das Quelldirectory gleichfalls als *Schablone* mit einem oder mehreren Jokern festlegen. In diesem Fall kopiert AmigaDOS jede Datei, die die vorgegebenen Kriterien der Schablone erfüllt. Schlagen Sie bei der Beschreibung des Befehles LIST nach. Dort finden Sie eine vollständige Beschreibung dieser Joker. Directory-Stufen können ebenso festgelegt werden wie Dateien oder solche Joker.

Aufgrund der in DOS 1.2 veränderten Disk-Organisation können Sie die Arbeitsgeschwindigkeit Ihres Rechners verbessern, wenn Sie ab und an Ihre Arbeitsdiskette mit dem Befehl COPY ALL auf eine andere leere Diskette kopieren und dann mit dieser weiterarbeiten.

Beispiel(e):

```
COPY datei1 TO :arbeit/datei2
```

kopiert die Datei »DATEI1 aus dem aktuellen Directory in das Directory ARBEIT und legt es dort als DATEI2 ab.

```
COPY system/#? TO DF1:backup
```

kopiert alle Files des Directory :SYSTEM in das Directory BACKUP auf der Diskette in Laufwerk DF1:.

COPY (Fortsetzung)

```
COPY DF0: TO DF1: ALL QUIET
```

erstellt eine Kopie der gesamten Diskette von Laufwerk 1 auf die Diskette in Laufwerk 2. Dabei werden die Namen der aktuell kopierten Dateien nicht auf dem Bildschirm ausgegeben.

```
COPY test-#? TO DF1:xyz
```

kopiert alle Dateien des aktuellen Directory, die mit »test-« beginnen, in das bereits auf der Diskette in Laufwerk 1 bestehende Directory XYZ.

```
COPY test.datei TO PRT:
```

gibt die Datei TEST.DATEI auf dem angeschlossenen Drucker aus.

```
COPY * TO CON:10/10/200/100/
```

Nach Eingabe dieses Befehls muß das Eingabefenster angeklickt werden. Dadurch wird es wieder zum Eingabefenster. Jedes Zeichen, das Sie nun eingeben, wird in das neu geöffnete Fenster kopiert. Mit <Ctrl>-\ wird das zweite Fenster wieder geschlossen.

```
COPY DF0:?/#? TO DF1: ALL
```

kopiert alle Files aus allen Unter-Directories in das Haupt-Directory auf der Diskette in DF1:.

Siehe auch: JOIN

DATE

Format: DATE [<datum>][<zeit>][TO|VER<name>]

Muster: DATE "DATUM,ZEIT,TO=VER/K"

Zweck: Stellen der Systemuhr.

Beschreibung: DATE zeigt das Datum und die Zeit im System und ermöglicht, sie neu zu stellen. DATE ohne Angabe von Parametern bringt die im System registrierte Datum- und Zeitanzeige und den Wochentag auf den Bildschirm. Die Zeit wird im 24-Stunden-Format dargestellt. Bitte verwechseln Sie die Systemuhr nicht mit der eventuell in Ihren Amiga (zum Beispiel Amiga 2000) eingebauten Hardware-Uhr. Für Informationen über die Beziehungen zwischen System- und Hardware-Uhr schauen Sie bitte einmal unter dem Befehl SETTIME beziehungsweise SETCLOCK nach.

DATE mit einer Datumsangabe dahinter setzt das Datum. Die Form für das Datum lautet TT-MMM-JJ, für T=Tag, M=Monat (in der englischen Schreibweise) und J=Jahr. Wenn das Datum schon gesetzt ist, kann es noch durch Eingeben eines Wochentagnamens oder durch die Bezeichnungen »Tomorrow« und »Yesterday« für »Morgen« und »Gestern« verändert werden.

DATE mit einer Zeitangabe ändert die Systemuhrzeit. Die Form der Zeitangabe ist SS:MM, für S=Stunden und M=Minuten. Die Eingabe muß immer vierstellig sein, zum Beispiel ist neun Minuten nach 24 Uhr 00:09. Zu beachten ist, daß AmigaDOS nur dann erkennt, daß die Zeit statt des Datums gemeint ist, wenn ein Doppelpunkt (:) verwendet wird. Hervorzuheben ist weiterhin, daß beide Anzeigen, Datum und Zeit, zusammen oder unabhänig voneinander und in jeder Reihenfolge eingegeben werden können. Dies ist möglich, da DATE sich nur auf die Zeit bezieht, wenn die Form SS:MM benutzt wird.

Haben Sie das Datum nicht gesetzt, dann setzt das Restart-Validation-Ablaufprogramm das Datum des Systems auf das Datum der zuletzt erzeugten Datei. Siehe auch Abschnitt 1.5 *Der Aktualisierungsprozeß.*

Um das Ziel der Nachprüfung genau zu bestimmen, müssen die entsprechenden Schlüsselwörter TO und VER benutzt werden. Das Ziel ist das Terminal, soweit Sie es nicht anderweitig bestimmen.

Beachten Sie bitte: Wenn Sie DATE eingeben, bevor der Vorgang der *Restart-Validation* abgeschlossen ist, erscheint die Zeit als ungestellt. Um in diesem Fall die Zeit zu setzen, können Sie entweder nochmals DATE verwenden oder Sie müssen auf das Ende der *Restart-Validation* warten.

DATE (Fortsetzung)

Beispiel(e):

```
DATE
```

zeigt die aktuelle Zeit der Systemuhr.

```
DATE 21-JAN-87
```

Setzt das aktuelle Datum auf den 21. Januar 1987. Die Zeit wurde nicht gestellt.

```
DATE TOMORROW
```

stellt das Datum einen Tag weiter.

```
DATE TO zeit
```

schreibt die Systemzeit in das File ZEIT.

```
DATE 10:50
```

setzt die Systemzeit auf 10.50 Uhr.

```
DATE 01-jan-02
```

setzt das Datum auf den 01. Januar 2002. (Das frühestmögliche Datum ist der 01. Januar 1978).

DELETE

Format: DELETE <name>[<name>*] [ALL]

Muster: DELETE ",,,,,,,,,ALL/S"

Zweck: Löschen einer oder mehrerer Dateien oder Directories.

Beschreibung: DELETE löscht bis zu zehn Dateien oder Directories. DELETE versucht, jede angegebene Datei zu löschen. Falls es eine Datei nicht löschen kann, wird eine Meldung auf dem Bildschirm ausgegeben, und AmigaDOS versucht, die nächste Datei in der Liste zu löschen. Es kann kein Directory gelöscht werden, solange sich noch eine Datei darin befindet.

Sie können auch einen Joker dazu verwenden, einen Dateinamen genauer zu bezeichnen. In der Beschreibung des Befehls LIST ist eine vollständige Übersicht der möglichen Joker zu finden. Diese Joker können eine Directory-Stufe ebenso gut eingrenzen wie Dateinamen. Falls Sie Joker eingegeben haben, werden alle Dateien, die in diese Schablone passen, gelöscht. Seien Sie also entsprechend vorsichtig damit!

Wird der Zusatz ALL und ein Directory-Name angehängt, dann löscht DELETE das angegebene Directory mit allen darin enthaltenen Unterdirectories und alle Dateien mit allen darin enthaltenen Directories und deren Unterdirectories.

Beispiel(e):

```
DELETE altfile
```

Löscht das File ALTFILE

```
DELETE arbeit/file1 arbeit/file2 arbeit
```

Löscht die Files FILE1 und FILE2 im Directory ARBEIT. Anschließend wird das leere Directory ARBEIT gelöscht.

```
DELETE t#?/#?(1|2)
```

Löscht alle Files, deren Namen mit »1« oder »2« enden und die in Directories stehen, deren Namen mit »t« beginnen. (Die Erklärungen zu den Zeichen # und ? finden Sie unter dem Befehl LIST später in diesem Kapitel.)

```
DELETE DF1:#? ALL
```

Löscht alle Files auf der Diskette in Laufwerk DF1:.

Siehe auch: DIR (Option DEL).

DIR

Format: DIR[<name>] [OPT A|I|AI]

Muster: DIR "DIR, OPT/K"

Zweck: Ausgabe des Directory-Inhaltes.

Beschreibung: DIR gibt die Dateien eines Directory in sortierter Form am Bildschirm aus. DIR kann Dateien aus Unterdirectories beinhalten, und DIR kann auch im interaktiven Dialog benutzt werden.

DIR ohne Parameter zeigt die Dateien im aktuellen Directory. DIR gefolgt von einem Directory-Namen zeigt die Dateien des bezeichneten Directory. Dabei werden zuerst stets alle Unterdirectories auf dem Bildschirm ausgegeben, gefolgt von einer sortierten, zweispaltigen Dateien-Liste. Diese Ausgabe können Sie mit <Ctrl>-C stoppen.

Mit dem Befehl »LIST DATEINAME« zeigt Ihnen Ihr Rechner an, ob eine solche Datei existiert. Geben Sie dagegen DIR DATEINAME ein, so teilt Ihnen Ihr Amiga mit, daß kein *Directory* dieses Namens existiert, selbst wenn eine *Datei* dieses Namens vorhanden ist.

Weitere Möglichkeiten bietet der OPT-Zusatz zusammen mit einem näher bestimmenden Parameter. Mit dem Parameter A schließen Sie alle Unterdirectories unter dem in der Liste gekennzeichneten ein, es wird also der Inhalt jedes Unterdirectories ausgegeben. Jede Dateien-Unterliste wird dabei etwas eingerückt.

Sollen nur Directory-Namen ausgegeben werden, so verwenden Sie die D-Option.

Die I-Option legt fest, daß DIR im interaktiven Dialog abläuft. In diesem Fall gibt der Rechner jede Datei und jedes Directory einzeln aus. Ein Fragezeichen dahinter signalisiert Ihnen, daß Ihr Amiga nun eine Eingabe Ihrerseits verlangt. Um den nächsten Namen zu erhalten, sollten Sie die Return-Taste betätigen. Möchten Sie den Dialog beenden, so geben Sie einfach »Q« ein. Um zur ursprünglichen Directory-Stufe zurückzukehren oder um abzubrechen, etwa weil man schon auf der Stufe des ursprünglichen Directory ist, gibt man »B« ein.

Ist der angezeigte Name ein Directory und sollen dessen Dateien und Unterdirectories ausgegeben werden, tippen Sie nur »E«. Sie verwenden »E« und »B«, um die verschiedenen Directory-Stufen auszuwählen. Mit dem Befehl »DEL« kann eine Datei oder sogar ein Directory gelöscht werden. Dies funktioniert aber nur dann, wenn das Directory keine Einträge mehr enthält. Mit dem Befehl »DEL« ist übrigens die Eingabe der drei Buchstaben D, E und L gemeint, nicht die DEL-Taste.

DIR (Fortsetzung)

Mit »T« wird die Datei am Bildschirm ausgegeben. <Ctrl>-C stoppt die Ausgabe und veranlaßt die Rückkehr in den interaktiven Dialog. Um die mögliche Antwort auf eine interaktive Frage zu finden, geben Sie »?« ein.

Beispiel(e):

```
DIR
```

gibt eine Liste aller Files des aktuellen Directory aus.

```
DIR DF0: OPT A
```

listet die gesamte Directory-Struktur der Diskette im ersten Laufwerk.

DISKCHANGE

Format: DISKCHANGE <drive>

Zweck: AmigaDOS wird mitgeteilt, daß Sie die Diskette in einem 5,25-Zoll-Laufwerk gewechselt haben.

Beschreibung: Rufen Sie DISKCHANGE auf, um AmigaDOS darüber zu informieren, daß Sie eine Diskette in Drive <drive> gewechselt haben. Dies ist nur bei 5,25-Zoll-Laufwerken notwendig, die keine automatische Rückmeldung vornehmen, sobald eine Diskette gewechselt wurde. Legen Sie also die neue Diskette ein und tippen Sie vor irgendwelchen anderen Aktivitäten:

```
DISKCHANGE df2:
```

DISKCOPY

Format: DISKCOPY [FROM] <disk> TO <disk> [NAME <name>]

Muster: DISKCOPY "FROM/A, TO/A/K, NAME/K"

Zweck: Kopieren einer Diskette.

Beschreibung: DISKCOPY erstellt eine Kopie des gesamten Inhalts der Diskette, die Sie mit FROM bezeichnet haben, und schreibt deren Inhalt auf die mit TO genauer bezeichnete Diskette. DISKCOPY formatiert die neue Diskette gleichzeitig während des Kopiervorgangs. Normalerweise wird der Befehl zur Erstellung von Sicherheitskopien (Backup) verwendet.

DISKCOPY arbeitet mit allen Diskformaten (auch Harddisks oder 5,25-Zoll-Disketten), die vorher mit dem Befehl MOUNT initialisiert wurden. Ziel und Quelle müssen allerdings dasselbe Format haben. Wollen Sie Informationen zwischen unterschiedlich genormten Disketten, wie einer 3,5- und einer 5,25-Zoll-Diskette, kopieren, müssen Sie den COPY-Befehl verwenden.

Nachdem Sie den Befehl eingegeben haben, fordert AmigaDOS Sie auf, die richtigen Disketten einzulegen. Zu diesem Zeitpunkt müssen Sie unbedingt die richtigen Ursprungs- und Zieldisketten einlegen!

Der Befehl kann auch zur Erstellung einer Kopie einer Diskette auf einem Einzellaufwerk verwendet werden. Wenn Sie Ursprung und Ziel mit der gleichen Device-Nummer belegen, liest das Programm soviel wie möglich von der Ursprungdiskette in den Speicher. Danach wird zum Einlegen der Zieldiskette in das Laufwerk aufgefordert und der Speicherinhalt auf die Zieldiskette kopiert. Dieser Ablauf wird so lange wiederholt wie notwendig.

Wird von Ihnen kein neuer Name für die Zieldiskette angegeben, so benennt DISKCOPY die neue Diskette mit dem Namen der alten. AmigaDOS kann zwischen zwei Disketten mit gleichlautenden Namen unterscheiden, denn auf jeder Diskette wird das Datum und die Zeit ihrer Erstellung vermerkt. DISKCOPY schreibt der neuen Diskette die aktuelle Datum- und Zeitangabe des Systems als Entstehungszeitpunkt in eine Systemspur.

DISKCOPY (Fortsetzung)

Unter DOS 1.2 existiert ein neuer Requester (Nachfragemeldung) für DISKCOPY:

```
Diskcopy:
Failed to open icon.library

Retry                    Cancel
```

Dieser Requester erscheint, wenn das Kommando DISKCOPY keinen Zugriff auf die Icon-Library auf der Workbench-Disk hat. Das passiert beispielsweise, wenn Sie unter CLI folgendes eingeben:

`COPY c:Diskcopy to RAM:`

(Entfernen Sie die Workbench-Disk)

`DISKCOPY`

(Jetzt erscheint ein Requester, der Sie auffordert, eine Diskette einzulegen, auf der sich das Directory LIBS: befindet; normalerweise die Workbench-Diskette.)

In diesem Fall legen Sie einfach die Workbench-Diskette ein.

Hinweis: Mit dem Befehl COPY TO RAM: können Sie einen Teil einer Diskette kopieren (siehe Abschnitt 1.6 dieses Handbuches).

Beispiel(e):

`DISKCOPY FROM DF0: TO DF1:`

kopiert die Diskette in Laufwerk 0 komplett auf die Diskette in Laufwerk 1.

`DISKCOPY FROM DF0: TO DF0:`

kopiert eine Diskette auf eine andere. Dabei wird nur Laufwerk 0 benutzt.

siehe auch:COPY

DISKDOCTOR

Format: DISCDOCTOR <dr>:

Zweck: Wiederherstellen zerstörter Disketten.

Beschreibung: Wenn Ihr Amiga eine zerstörte Diskette in einem Laufwerk vorfindet, so wird das sofort gemeldet (eine Diskette kann zum Beispiel dann zerstört werden, wenn Sie sie aus dem Laufwerk nehmen, obwohl die Laufwerks-LED noch leuchtet). DISKDOCTOR stellt nun so viele Files wieder her, wie es vermag. Wenn Sie eine Diskette mit diesem Befehl bearbeitet haben, dann sollten Sie alle darauf befindlichen Files schleunigst auf eine »gesunde« Diskette kopieren (nicht mit DISKCOPY, sondern mit COPY!) und die beschädigte Diskette neu formatieren.

Beispiel(e):

AmigaDOS hat eine zerstörte Diskette identifiziert, falls einer der folgenden Requester auf dem Bildschirm erscheint:

```
Volume
Textfiles
is not validated
```

oder:

```
Error validating disks
Disk is unreadable
```

Nachdem Sie noch ein paar Leseversuche unternommen haben, geben Sie ein (die Diskette befindet sich in DF1:):

```
DISKDOCTOR DF1:
```

Wenn der Befehl seine Arbeit getan hat, dann erscheint die Meldung:

```
Now copy files required to a new disk and reformat this disk.
```

ECHO

Format: ECHO <zeichenkette>

Muster: ECHO " "

Zweck: Ein Argument ausgeben.

Beschreibung: ECHO gibt den Text aus, den Sie als Argument übergeben. Es schreibt diesen Text in die aktuelle Ausgabe-Richtung, also eine Datei oder ein Device (oder auf den Bildschirm). Dies ist normalerweise nur innerhalb einer Befehlsfolge oder als Teil eines RUN-Befehls sinnvoll. Wird das Argument falsch eingegeben, wird eine Fehlermeldung ausgegeben.

Beispiel(e):

```
RUN COPY :arbeit/program TO DF1:arbeit ALL +
ECHO "Kopie fertig"
```

erzeugt ein neues CLI, kopiert in diesem CLI das angegebene Directory und gibt anschließend die Meldung KOPIE FERTIG aus. Dieses CLI läuft als Hintergrund-Task ab.

Erstellen Sie nun folgendes EXECUTE-File mit dem Namen FILEKOPIE (die Files ABC und XYZ werden auch benötigt!):

```
ECHO "Start des EXECUTE-Files"
COPY DF1:ABC TO RAM: ABC
COPY DF1:XYZ TO RAM:XYZ
ECHO "Entnimm die Diskette in DF1:"
ECHO "Lege eine neue Diskette in DF1:"
WAIT 10 SECS
COPY RAM:ABC TO DF1:ABC
COPY RAM:XYZ TO DF1:XYZ
ECHO "FERTIG!"
```

dann führt der Befehl

```
EXECUTE filekopie
```

zur Kopie der angegebenen Files auf die RAM-Disk und von dort auf eine neue Diskette.

ED

Format: ED [FROM] <name> [SIZE<n>]

Muster: ED "FROM/A, SIZE"

Zweck: Einen Text bearbeiten.

Beschreibung: ED ist ein Bildschirmeditor und editiert Textdateien. Er kann wahlweise statt des Zeileneditors EDIT benutzt werden. Die FROM-Datei, sie wird nach dem Befehl ED angegeben, wird in den Speicher gelesen. Danach akzeptiert ED Editierbefehle. Wenn die FROM-Datei nicht auf der Diskette existiert, wird sie von AmigaDOS erzeugt.

Da die Datei in den Speicher gelesen wird, gibt es eine willkürlich festgelegte Grenze für die maximale Größe einer Datei, die mit ED editiert werden kann. Falls nicht anders angegeben, beträgt die Größe des Editierspeichers 40.000 Bytes. Diese Größe reicht gewöhnlich für die meisten Dateien aus. Für den Fall, daß die Größe des Arbeitsspeicher geändert werden muß, geben Sie einen geeigneten Wert nach dem Zusatz SIZE ein.

Eine ausführliche Erklärung von ED finden Sie in Kapitel 3.

Beispiel(e):

```
ED arbeit/file
```

ruft den Editor auf und lädt das File FILE aus dem Directory ARBEIT, falls dieses File nicht existiert, wird es neu angelegt.

```
ED riesenfile SIZE 50000
```

liest das File RIESENFILE in den Editor. Der zugewiesene Speicherplatz beträgt 50 Kbyte.

EDIT

Format: EDIT [FROM] <name> [[TO] <name>] [WITH<name>] [VER<name>] [OPT<option>]

Muster: **EDIT "FROM/A, TO, WITH/K, VER/K, OPT/K"**

Zweck: Textfiles editieren.

Beschreibung: EDIT ist ein Zeileneditor und dient zum Editieren von Textdateien, das heißt, mit ihm kann eine sequentielle Datei Zeile für Zeile überarbeitet werden. Wird eine Datei TO genauer bezeichnet, so kopiert er von der in der FROM-Position bezeichneten Ursprungsdatei zu der hinter TO angegebenen Zieldatei. Sobald Sie das Editieren beendet haben, befindet sich in der Zieldatei das editierte Ergebnis, während die Ursprungsdatei unverändert bleibt. Geben Sie TO dagegen nicht an, so schreibt EDIT in eine vorläufige Datei. Hängen Sie hinter dem Befehl EDIT die Argumente »Q« oder »W« an, dann benennt EDIT diese vorläufige Datei mit dem Namen der Ursprungsdatei, nachdem es zuerst die alte Version der Ursprungsdatei unter dem Namen :T/EDIT-BACKUP abgespeichert hat. Haben Sie EDIT statt dessen den Befehl STOP gegeben, wird die Ursprungsdatei nicht verändert.

EDIT liest die Befehle aus der aktuellen Eingaberichtung oder, falls angegeben, aus einer WITH-Datei.

Das Programm EDIT schickt Editor-Mitteilungen und Ausgabebestätigungen an die Datei, die Sie mit VER genauer bezeichnen. Lassen Sie VER fort, werden diese Meldungen an das Terminal gesandt.

OPT legt die Rahmenbedingungen fest: *Pn* begrenzt die maximale Anzahl an vorhergehenden Zeilen auf »n«. *Wn* legt die maximale Zeilenlänge fest. Die voreingestellten Werte sind P40W120.

Hinweis: Die Befehle »<« und »>« zur Umleitung der Ein- und Ausgaberichtung sind in EDIT nicht erlaubt. In Kapitel 4 wird der Zeilen-Editor EDIT näher erläutert.

Beispiel(e):

```
EDIT arbeit/file
```

ermöglicht die Bearbeitung des Files FILE im Directory ARBEIT. Nach Ende der Bearbeitung wird die alte Version von ARBEIT/FILE unter dem Namen ARBEIT/FILE in :T/EDIT-BACKUP gespeichert.

EDIT (Fortsetzung)

```
EDIT arbeit/file TO arbeit/neufile
```

lädt das File ARBEIT/FILE zur Bearbeitung. Das neue File wird unter dem Namen ARBEIT/NEUFILE gespeichert.

```
EDIT arbeit/file WITH editfile/O VER prt:
```

das File ARBEIT/FILE wird mit den in EDITFILE/O gespeicherten Befehlen bearbeitet. Alle Meldungen des Editors werden auf dem Drucker ausgegeben.

ENDCLI

Format: ENDCLI

Muster: ENDCLI

Zweck: Ein CLI-Fenster schließen.

Beschreibung: ENDCLI beendet einen CLI-Prozeß. AmigaDOS erlaubt ENDCLI nur als interaktiven Befehl (also nicht in EXECUTE-Dateien).

ENDCLI sollten Sie unter keinen Umständen außerhalb eines CLI, das mit dem Befehl NEWCLI aufgebaut wurde, verwenden. Wenn das erste CLI (TASK 1) beendet ist und vorher kein weiteres durch den Befehl NEWCLI aufgebaut wurde, beendet ENDCLI die Arbeit mit AmigaDOS.

Für den Befehl ENDCLI gibt es keine Argumente. Wurde das CLI beim Systemstart direkt erstellt (also nicht über die Workbench, sondern durch Abändern der Startup-Sequenz), erzeugt der Befehl ENDCLI im letzten Task automatisch ein neues CLI.

Beispiel(e):

```
NEWCLI
DIR
ENDCLI
```

eröffnet ein neues CLI, listet das aktuelle Directory auf und schließt das CLI wieder.

EXECUTE

Format: EXECUTE <anweisungsfile> [<argumente>*]

Muster: EXECUTE "anweisungsfile","argumente"

Zweck: Ausführen einer Befehlsdatei mit Argumentenübergabe.

Beschreibung: EXECUTE dient normalerweise zum Einsparen von Tipparbeit. Eine Befehlsdatei enthält Befehle, die durch das CLI ausgeführt werden (also alle Befehle, die wir hier besprechen). AmigaDOS führt diese Befehle einen nach dem anderen aus, genau in der Reihenfolge, wie sie in das Befehlsfile getippt wurden. Normalerweise reagiert der Rechner haargenau so, als wenn Sie die Befehle direkt ins CLI-Fenster eingegeben hätten. Wenn durch die Befehle allerdings ein neues CLI-Fenster entsteht, kann es sein, daß das Ergebnis nicht mit dem einer Tastatureingabe übereinstimmt.

EXECUTE erzeugt übrigens ein Directory mit dem Namen SYS:T, sofern es noch nicht existiert.

EXECUTE kann auch verwendet werden, um Parameter (Werte) in die einzelnen EXECUTE-Befehle einzusetzen, wobei Sie verschiedene Namen als Parameter angeben können. Bevor die Befehlsdatei ausgeführt wird, vergleicht AmigaDOS die Namen der Parameter in der Befehlsdatei mit denen, die Sie nach dem Befehl EXECUTE angegeben haben. In jedem folgenden Vergleich wird AmigaDOS die angegebenen Werte statt der entsprechenden Namen der Parameter verwenden (es wird praktisch ein Suchen und Ersetzen wie in einer guten Textverarbeitung ausgeführt).

Parameter können Default-Werte annehmen, die AmigaDOS verwendet, wenn Sie sie im EXECUTE-Befehl nicht angegeben haben. Haben Sie hinter EXECUTE keine Parameter angegeben, obwohl die Befehlsdatei solche erwartet, und sind Default-Werte eingestellt, dann ist der Wert des jeweiligen Parameters nicht belegt, und es wird nichts dafür eingesetzt.

Um die Parameter-Technik einzusetzen, müssen Sie diverse Sonderbefehle in der Befehlsdatei angeben. Um innerhalb dieser Datei anzugeben, daß nun ein Sonderbefehl folgt, wird die entsprechende Zeile mit einem Punkt (.) begonnen.

EXECUTE (Fortsetzung)

Die möglichen Sonderbefehle lauten wie folgt:

.KEY		Kennzeichnung, daß nun die Definition der verschiedenen Argumente folgt. Abgekürzt mit .K
.DOT	ch	Ändert das Punkt-Zeichen (anfänglich ».«) in ch.
.BRA	ch	Ändert das Kleiner-Zeichen (ursprünglich »<«) in ch.
.KET	ch	Ändert das Größer-Zeichen (ursprünglich »>«) in ch.
.DOLLAR	ch	Ändert das Default-Zeichen (ursprünglich »$«) in ch, abgekürzt als .DOL
.DEF	Schlüsselwort	Gibt Ersatzwert an Parameter
.<Leerzeichen>	Kommentarzeile	
.<Neue Zeile>	Leerzeile	

Vor der Ausführung durchsucht AmigaDOS den Inhalt der Datei auf jeden Begriff, der von den Kleiner- und Größer-Zeichen »<« und »>« eingeschlossen wird. Solche Begriffe können ein Schlüsselwort (also den Namen eines Übergabeparameters) oder ein Schlüsselwort und ein Ersatzwort (= Default-Wert) beinhalten, für den Fall, daß der jeweilige Parameter nicht angegeben wurde. Um das Schlüsselwort und sein Ersatzwort, falls vorhanden, zu trennen, wird das Dollar-Zeichen »$« verwendet. Auf diese Weise ersetzt AmigaDOS den Ausdruck »<TIER>« durch den Wert, den Sie hinter EXECUTE für das Schlüsselwort TIER angegeben haben. Das gleiche passiert mit dem Ausdruck <TIER$KATZE>. Haben Sie für TIER jedoch keinen Wert übergeben, so wird als Default-Wert einfach KATZE eingesetzt.

Eine EXECUTE-Datei kann Punkt-Befehle nur verwenden, wenn in der ersten Zeile ein Punkt-Befehl steht! Das CLI untersucht die erste Zeile, und wenn diese mit einem Punkt-Befehl (beispielsweise einem Kommentar) beginnt, durchsucht das CLI die Datei auf eingesetzte Parameter und erzeugt eine vorläufige Datei in dem :T-Directory. Wenn dagegen die Datei nicht mit einem Punkt-Befehl beginnt, wird angenommen, daß sich keine Punkt-Befehle in ihr befinden und damit auch, daß keine Parameter ersetzt werden sollen. Für den Fall, daß sich kein Punkt-Befehl in der ersten Zeile befindet, beginnt CLI die Datei auszuführen, ohne sie zunächst in das Directory :T zu kopieren, was sonst passieren würde.

Zu beachten ist, daß Kommentare auch in Execute-Dateien möglich sind, indem Sie das Kommentar-Zeichen des CLI, das Semikolon (;) setzen. Sollten das Einsetzen von Parametern und die Punkt-Befehle nicht unbedingt nötig sein, wäre es besser, Sie ließen sie fort. Sie ersparen dem Rechner zusätzliche (zeitraubende) Zugriffe auf die Diskette.

EXECUTE (Fortsetzung)

AmigaDOS stellt eine Reihe von Befehlen zur Verfügung, die nur in Dateien mit Befehls-Sequenzen sinnvoll sind. Dazu gehören IF, SKIP, LAB und QUIT. Diese können in Befehlssdateien eingebettet werden. Wichtig ist, daß Sie auch weitere EXECUTE-Dateien innerhalb einer EXECUTE-Datei aufrufen können. Das heißt, EXECUTE-Dateien können EXECUTE-Befehle beinhalten. Um die Ausführung einer Befehlsdatei zu unterbrechen, muß <Ctrl>-D gedrückt werden. Mit <Ctrl>-C wird die Ausführung ineinander verschachtelter EXECUTE-Dateien vorzeitig beendet. <Ctrl>-D beendet nur die jeweils aktuelle Datei. Die Control-Tasten können natürlich auch mit BREAK simuliert werden.

Beispiel(e):

Angenommen, das File LIST sieht so aus:

```
.K filename/A
RUN COPY <filename> TO PRT:+
ECHO "<filename> ausgedruckt!"
```

Dann ergibt der Befehl

```
EXECUTE list testprogramm
```

das gleiche Ergebnis wie die direkte Eingabe von:

```
RUN COPY testprogramm TO PRT:+
ECHO "testprogramm ausgedruckt"
```

Noch ein etwas umfangreicheres Beispiel. Das Befehlsfile heißt diesmal ANZEIGER:

```
.KEY name/A
IF EXISTS <name>
TYPE <name> OPT n
ELSE
ECHO "<name> steht nicht in diesem Directory"
ENDIF
```

Der Befehl

```
RUN EXECUTE anzeiger testfile
```

gibt das File TESTFILE mit Zeilennummern aus, wenn es im aktuellen Directory steht, ansonsten erscheint die Meldung TESTFILE STEHT NICHT IM AKTUELLEN DIRECTORY.

Siehe auch: IF, SKIP, FAILAT, LAB, ECHO, RUN, QUIT

EXECUTE (Fortsetzung)

Zusätzliche Beispiele zum Befehl EXECUTE

Beispiel 1:

Parameterübergabe durch Schlüsselwörter und/oder durch die Position.

Der Ausdruck .KEY (oder abgekürzt auch .K) berücksichtigt sowohl die unterschiedlichen Namen der Schlüsselwörter als auch die Reihenfolge der Argumente. Er teilt dem Befehl EXECUTE mit, wie viele Argumente hinter EXECUTE erwartet werden müssen und wie sie zu interpretieren sind. Mit anderen Worten ist der .KEY-Befehl mit allem, was dahinter angegeben wird, eine Art Eingabe-Schablone für die Parameter, die Sie hinter dem EXECUTE-Befehl angeben sollten. Es ist nur eine .KEY-Zuweisung pro Befehlsdatei erlaubt. Wird von der Möglichkeit Gebrauch gemacht, so muß sie in der ersten Zeile der gesamten Befehlsdatei stehen.

Nehmen wir also an, unser erstes Befehlsfile mit dem Namen DEMO1 enthält als erste Zeile folgende .KEY-Zuweisung:

```
.KEY blitz, pfanne
```

AmigaDOS weiß nun bei der Ausführung des Files, daß zwei Argumente zu übernehmen sind, nämlich <blitz> und <pfanne>. Angenommen, Sie geben nun in Ihrem CLI-Fenster interaktiv folgende Befehlszeile ein (beachten Sie, daß das Befehlsfile hier den Namen DEMO1 besitzt):

```
EXECUTE DEMO1 pfanne einname blitz anderername
```

dann wird der Name »einname« dem Schlüsselwort PFANNE und der Name ANDERERNAME dem Schlüsselwort BLITZ zugeordnet.

Durch Einhalten der in dem ».KEY«-Befehl gegebenen Reihenfolge der Parameter jedoch können Sie sich die Angabe der Schlüsselworte einsparen. Folgende Befehlszeile erfüllt den gleichen Zweck wie die oben gezeigte:

```
EXECUTE DEMO1 anderername einname
```

dies nur, weil sich die Eingabe der Argumente an das in der ».KEY«-Befehlszeile gegebene Muster hält.

Beide Formen der Parameterübergabe können auch gemischt angewendet werden. Der nächste ».KEY«-Befehl bestimmt zum Beispiel vier Argumente:

```
.KEY wort1, wort 2, wort3, wort4
```

EXECUTE (Fortsetzung)

Der folgende EXECUTE-Befehl führt dann durchaus zum richtigen Ergebnis (unsere Befehlsdatei heißt hier DEMO1):

```
EXECUTE DEMO2 wort3 ccc wort1 aaa bbb ddd
```

AmigaDOS ordnet den Wert ccc vereinbarungsgemäß dem Parameter <wort3> zu. <wort1> erhält den Wert aaa. Der nächste unbelegte Parameter ist also <wort2> und dem wird gemäß Vereinbarung der Wert bbb zugewiesen. Der letzte Wert ddd bleibt dann dem letzten freien Parameter <wort4>.

Jeder einzelne Parameter kann noch zusätzlich konditioniert werden:

```
.KEY name1/a, name2/a, name3, name4/k
```

Das »/a« bedeutet, daß den Parametern <name1> und <name2> in jedem Fall Werte übergeben werden müssen. Werte für die Parameter <name3> und <name4> können übergeben werden, sind aber nicht unbedingt erforderlich. Andererseits legt das »/k« von Parameter <wort4> fest, daß die Angabe des Schlüsselwortes (hier name4) zwingend ist.

```
EXECUTE demo3 eins zwei drei name4 vier
```

ist eine für das obige Befehlsfile gültige Wertzuweisung. Ein Fehler bei der Parameter-Übergabe bricht die Ausführung des EXECUTE-Files mit einer Fehlermeldung ab.

Hier noch ein Beispiel für die Option »/k«:

Ein EXECUTE-File COMPILIERE soll ein Textfile TEXTFILE in den Binärcode des Rechners übertragen und den Binärcode in ein File SCHREIBFILE schreiben, wenn dies gewünscht wird. Der ».KEY«-Befehl könnte so aussehen:

```
.k compilierwas/A, schreibes/K
```

Geben Sie dann diese Befehlszeile ein:

```
EXECUTE COMPILIERE textfile schreibes schreibfile
```

Nur so wird der Befehl korrekt ausgeführt, da die optionale Ausgabe des TEXTFILE in die Datei SCHREIBFILE mit Schlüsselwort und Parameter angefordert wurde.

EXECUTE (Fortsetzung)

Beispiel 2:

Wertzuweisung an Default-Parameter und Umdefinition der Klammerzeichen

Mit dem Befehl .DEF wird einem Parameter, der im Aufruf des EXECUTE-Files von Ihnen keinen Wert zugewiesen bekommt, ein *Ersatzwert* zugewiesen. Dieser Ersatzwert muß zugewiesen werden, bevor der Parameter das erste Mal verwendet wird. Der Ersatzwert wird in Anführungszeichen ("") gesetzt oder – durch das Dollarzeichen ($) getrennt – direkt hinter dem zu ersetzenden Schlüsselwort im EXECUTE-File eingegeben. Bei dieser Möglichkeit muß der Ersatzwert bei jeder Verwendung dieses Schlüsselwortes neu eingegeben werden.

Die folgende Zeile zeigt Möglichkeit zwei. Hier wird also speziell für eine einzige Stelle im EXECUTE-File ein Default-Wert angegeben:

```
ECHO "<wort1$ersatzwort1> ist der Ersatz für wort1."
```

Die globale Definition eines Ersatzwortes sieht so aus (sie gilt also für das gesamte EXECUTE-File):

```
.DEF wort1 "ersatzwort1"
```

Lassen Sie nun folgende Zeile durch EXECUTE ausführen:

```
ECHO "<wort1> ist der Ersatz für wort1."
```

Die Ausgabe beider Versionen wird lauten:

```
ersatzwort1 ist der Ersatz für wort1.
```

Eine zweite Definition eines Ersatzwertes für den gleichen Parameter wird ignoriert!

Definition anderer Klammerzeichen:

Wann immer EXECUTE ein Schlüsselwort in spitzen Klammern vorfindet, sucht es den passenden Wert dazu. Ein nicht definierter Parameter ohne Default-Zuweisung bekommt einen Leerstring zugewiesen. Möchten Sie jetzt die spitzen Klammern aber innerhalb eines Strings als Zeichen verwenden, gibt das Probleme – aber auch dafür eine Lösung:

EXECUTE (Fortsetzung)

Mit den Befehlen .BRA und .KET können Ersatzzeichen für die spitzen Klammern definiert werden. Das folgende Befehlsfile »TEST« nutzt diese Möglichkeit:

```
.KEY wort1
ECHO "Diese Zeile druckt KEINE <eckigen> Klammern!"
.BRA {
.KET }
ECHO "Diese Zeile DRUCKT <eckige> Klammern!"
ECHO "Das Ersatzwort ist {wort1}."
```

Mit dem folgenden EXECUTE-Befehl:

```
EXECUTE TEST ersatzwort
```

ergibt das dann folgende Ausgabe am Bildschirm:

```
Diese Zeile druckt KEINE Klammern!
Diese Zeile DRUCKT <eckige> Klammern!
Das Ersatzwort ist ersatzwort.
```

Der erste ECHO-Befehl suchte nach einem Argument für den Parameter »eckige«, konnte keines finden und teilte dem Parameter einen Leerstring zu. Dann wurden die geschweiften Klammern als Ersatz für die eckigen Klammern definiert, die ihrerseits damit zu normal druckbaren Zeichen wurden, was der zweite ECHO-Befehl bewies. Der dritte ECHO-Befehl beweist die Funktion der neuen Parameter-Kennung.

Beispiel 3:

Programmstrukturen in EXECUTE-Files

Der Befehl IF (zu deutsch: falls) erlaubt die Ausführung eines weiteren Befehls in Abhängigkeit einer erfüllten oder nicht erfüllten Bedingung. Mit diesem Befehl können zum Beispiel zwei Strings auf Gleichheit oder die Existenz eines Strings überprüft werden. Der Befehl ELSE schreibt den weiteren Ablauf des Programmes vor, wenn die IF-Bedingungen nicht erfüllt sind. Der Befehl ENDIF beendet den Block der IF-Befehle. Sie kennen das sicher aus der Programmiersprache BASIC und vielen anderen Sprachen.

Im folgenden Beispiel wird zusätzlich der Befehl SKIP (Sprung) verwendet. Dieser Befehl erlaubt nur Sprünge vorwärts zu einer später mit LAB definierten Programmposition.

EXECUTE (Fortsetzung)

Hier das kleine Beispiel:

```
IF "<wort1>" EQ "testwort"
SKIP sprung
ENDIF
SKIP fertig
LAB sprung
ECHO "Bedingung erfüllt"
LAB fertig
```

Schließen Sie die IF-Parameter immer in Anführungszeichen ein! Vergessen Sie im EXECUTE-Befehl, den Parameter zu übergeben, lautet unsere erste Zeile:

```
IF "" EQ "testwort"
```

Das ist zwar ein sinnloser Befehl, da unsere Bedingung nie erfüllt wird (ein Leerstring kann niemals ein Wort enthalten!), aber ein zulässiger.

Haben Sie die Anführungszeichen nicht eingegeben, lautet die erste Zeile:

```
IF EQ "testwort"
```

darauf hat AmigaDOS nur eine Antwort:

`EQ must have two arguments` (EQ braucht zwei Argumente)
`IF failed returncode 20` (Vergleich nicht möglich, Fehlercode 20)

Im IF-Befehl ist zusätzlich das Wort NOT (nicht) erlaubt. Durch den Zusatz von NOT wird das Abfrageergebnis negiert (aus wahr wird falsch und umgekehrt), das heißt, die Befehle nach IF werden in unserem Beispiel ausgeführt. Zum Beispiel:

```
IF NOT EXISTS <von>
```

führt die Befehle nach IF aus, wenn eine Datei »von« nicht existiert.

In der Zeile mit dem IF-Befehl darf neben den Testbedingungen kein weiterer Befehl stehen. Folgende Zeile ist deshalb nicht zulässig:

```
IF "<etwas>" EQ "wahr" SKIP fertig
```

Die korrekte Form lautet:

```
IF "<etwas>" EQ "wahr"
SKIP fertig
ENDIF
```

wobei die Konstante »wahr« nicht in Anführungszeichen stehen muß.

EXECUTE (Fortsetzung)

In unserem nächsten EXECUTE-File zeigen wir noch die Verschachtelung von IF-Befehlen. So können innerhalb eines Befehlsfiles mehrere Bedingungen auf »richtig« oder »falsch« überprüft werden. Tabulatoren erhöhen die Lesbarkeit eines langen Befehlsfiles, machen Sie also davon Gebrauch, wenn ein File öfter verwendet werden soll.

Das nachfolgende EXECUTE-File namens KP (für »*kopiere*«) kopiert ein File. Als Befehle werden verwendet: IF.....[ELSE]....ENDIF, LAB und SKIP. Das File selbst tut nichts anderes als der Befehl COPY, demonstriert aber sinnvolle Strukturen für Befehlsfiles.

```
.KEY von,zu,o      ; Parameter zuordnen
IF "<von>" EQ ""   ; FROM-File eingegeben?
SKIP anweisung     ; Kein File, dann Befehl ausgeben
ENDIF
IF "<zu>" EQ ""    ; TO-File eingegeben?
SKIP anweisung     ; Kein File, dann Anweisung ausgeben
ENDIF
IF NOT EXISTS <von> ; FROM-File vorhanden?
ECHO "Das Ursprungsfile existiert nicht,"
ECHO "erst mit LIST oder DIR nach dem File suchen,"
ECHO "dann nochmal starten."
SKIP fertig  ; Es kann jederzeit aus einer Schleife
             ; herausgesprungen werden.
ENDIF
IF EXISTS <zu>  ; Gibt es das TO-File schon?
  IF "<o>" EQ "Ü" ; falls ja, wurde »Ü«berschreiben
                  ; eingegeben?
    ECHO "Das File <zu> wird von einer Kopie des Files"
    ECHO "<von> überschrieben!" ; und die Meldung ausgegeben.
  ELSE
    ECHO "Dieser Befehl überschreibt ein bestehendes"
    ECHO "File nur, wenn der Parameter "Ü" eingegeben wurde"
    ECHO "Ausführung abgebrochen!"
    SKIP anweisung          ; Sprung zur Anweisung
  ENDIF
ELSE
  COPY FROM <von> TO <zu>; File wird kopiert
                         ;TO-File wird neu errichtet.
  ECHO "Kopiere <von> als <zu>."
ENDIF
```

EXECUTE (Fortsetzung)

```
SKIP fertig      ; Anweisung soll nur ausgegeben werden,
                 ; wenn ein Fehler gemacht wurde.
LAB anweisung
ECHO "Anweisung zum EXECUTE-File kp:"
ECHO "Nur folgende Kopierbefehle werden ausgeführt:"
ECHO "x kp ursprungfile zielfile"
ECHO "x kp ursprungfile zielfile Ü"
ECHO "wobei x für EXECUTE, kp für dieses Befehlsfile und "
ECHO "Ü für überschreiben eines bestehenden Files steht."
LAB fertig
```

Sie sehen, wie kompliziert ein EXECUTE-File werden kann. Berücksichtigen Sie stets, daß jeder einzelne Befehl (auch IF oder ENDIF und so weiter) aus dem C-Ordner Ihrer Diskette geladen werden muß, bevor AmigaDOS ihn ausführen kann. Bei einem EXECUTE-File dieser Länge kann das schon eine Weile dauern. Deshalb empfiehlt es sich in vielen Fällen, den C-Ordner – oder Teile davon – in die RAM-Disk zu kopieren (ASSIGN oder PATH nicht vergessen!).

Beispiel 4:

Schleifenprogrammierung in EXECUTE-Files

Der Befehl SKIP erlaubt nur Sprünge zu später im Befehlsfile stehenden Befehlen. Sprünge zu bereits vorher abgearbeiteten Befehlen sind nur durch interaktives Aufrufen von EXECUTE möglich. Das folgende kleine Beispiel soll dies demonstrieren.

```
.KEYpar1,par2,zähler,marke
FAILAT 20
IF NOT "<marke>" EQ ""  ; Marke definiert?
  SKIP <marke>          ; dann Sprung zu Marke
ENDIF
ECHO "Diese Mitteilung erscheint nur zu Beginn. (<par1> <par2>)"
LAB 1.schleife
IF "<zähler>" EQ "III"            ; Fertig mit der Schleife?
 SKIP schleifenende-<marke>"      ; ja, dann raus
ENDIF
ECHO "Schleife <zähler>I."
EXECUTE schleife <par1> <par2> <zähler>I 1.schleife
                                        ; File erneut aufrufen
```

EXECUTE (Fortsetzung)

```
LAB schleifenende-<marke>
IF NOT "<zähler>" EQ ""
  SKIP ende
ENDIF              ; Ende der Schleife
ECHO "Diese Mitteilung erscheint nur zum Schluß. (<par1>,<par2>)"
LAB ende
```

Dieses File muß mit EXECUTE SCHLEIFE wert1 wert2 ohne weitere Parameter-Übergabe aufgerufen werden! Es schreibt dann folgende Meldungen auf den Bildschirm:

```
Diese Mitteilung erscheint nur zu Beginn. alpha beta
Schleife I
Schleife II
Schleife III
Diese Mitteilung erscheint nur zum Schluß. alpha beta
```

FAILAT

Format: FAILAT <n>

Muster: FAILAT "RETURNCODELIMIT"

Zweck: Verändern des Abbruchlimits bei Fehlermeldungen.

Beschreibung: FAILAT instruiert eine Befehlssequenz, die Ausführung abzubrechen, wenn eine Fehlermeldung größer oder gleich einer angegebenen Zahl ist.

Befehle, die nicht ausgeführt werden konnten, zeigen dies an, indem sie einen Fehlercode zurückmelden. Ein Fehlercode größer als null zeigt an, daß ein Fehler der jeweiligen Art gefunden wurde. Ein Fehlercode größer oder gleich einem bestimmten Wert (der sogenannten Fehlergrenze) bricht eine Folge nicht interaktiver Befehle ab (zum Beispiel in einer EXECUTE-Datei oder die Kommandos, die Sie nach einem RUN eingeben). Der Fehlercode zeigt dabei an, wie schwerwiegend der Fehler war. Er nimmt in der Regel die Werte 5, 10 oder 20 an. Der Befehl FAILAT wird nun benutzt, um die Abbruchgrenze (normalerweise 10) zu verändern. Wird der Wert vergrößert, werden die Fehler, die eine Nummer kleiner als die Abbruchgrenze besitzen, als nicht so schwerwiegend behandelt, und die Ausführung nachfolgender Befehle wird trotz Fehler fortgesetzt.

Das Argument muß eine positive Zahl sein. Die voreingestellte Größe zum Abbruch der Befehlssequenz ist der Wert 10. FAILAT kann vor Befehlen wie IF verwendet werden, um zu testen, ob ein Befehl abgebrochen wurde. Anderenfalls wird die Befehlsfolge beendet, bevor der Befehl IF ausgeführt wurde.

Wird das Argument weggelassen, zeigt FAILAT den augenblicklich eingestellten Wert an.

Beispiel(e):

Die nachfolgende Befehlssequenz wird nur abgebrochen, wenn ein Fehler-Return-Code >= 25 auftritt.

```
FAILAT 25
```

Siehe auch: IF, EXECUTE, RUN, QUIT

FAULT

Format: FAULT [<n>*]

Muster: FAULT ",,,,,,,,,"

Zweck: Ausgabe von Fehlermeldungen.

Beschreibung: AmigaDOS liefert die Fehlermeldungen, deren Codes eingegeben wurden, im Klartext. Bis zu zehn Fehlercodes können nacheinander eingegeben werden. Der Befehl erspart Ihnen also lästiges Nachschlagen.

Beispiel(e):

```
FAULT 222
```

Schreibt auf den Bildschirm:

```
FAULT 222: file is protected from deletion
```

```
FAULT 221 103 121 218
```

ergibt folgende Meldung:

```
Fault 221: disk full
Fault 103: insufficient free store
Fault 121: file is not an object module
Fault 218: device (or volume) not mounted
```

FILENOTE

Format: FILENOTE [FILE] <file> COMMENT <kommentar>

Muster: FILENOTE "FILE/A,COMMENT/K"

Zweck: Anhängen einer Dateinotiz.

Beschreibung: FILENOTE ordnet einer gekennzeichneten Datei einen Kommentar zu. Das Schlüsselwort COMMENT leitet einen bis zu 80 Zeichen langen Kommentar ein. Ein Kommentar kann mehr als ein Wort umfassen, das heißt, er darf Leerzeichen zwischen den Worten oder Zeichen enthalten. In diesem Fall muß der Kommentar zwischen Anführungsstrichen (") stehen.

Ein Kommentar ist mit einer speziellen Datei verknüpft. Untersuchen Sie diese Datei mit dem Befehl LIST, erscheint eine Zeile tiefer der Kommentar:

```
prog      30 rwed Today 11:07:33
: version 3.2 - 23. März 85
```

Eröffnen Sie eine neue Datei, so ist ihr kein Kommentar angefügt. Wenn Sie eine bestehende Datei, der ein Kommentar zugeordnet ist, überschreiben, so wird der Kommentar beibehalten, selbst wenn der Inhalt der Datei geändert wurde. Der Befehl COPY kopiert nur die Datei. Wird eine Datei mit Kommentar kopiert, enthält die Kopie keinen Kommentar.

Beispiel(e):

```
FILENOTE programm2 COMMENT "Version 3.3 vom 30. März 85"
```

hängt der Datei PROGRAMM2 den Kommentar »Version 3.3 vom 30. März 85« an.

Siehe auch: LIST, COPY

FORMAT

Format: FORMAT DRIVE <drivename> NAME <diskettenname> [NOICONS]

Muster: FORMAT "DRIVE/A/K, NAME/A/K, NOICONS"

Zweck: Erstellen einer für den Amiga verwertbaren Diskette.

Beschreibung: FORMAT formatiert eine neue Diskette durch Anlegen von Spuren und Sektoren, die von AmigaDOS benötigt werden. Zugleich werden Directory und Kennzeichnung für belegte und unbelegte Blöcke angelegt. Nur eine formatierte Diskette erhält den gewünschten Namen und kann nun beschrieben und wieder gelesen werden. Zu beachten ist, daß Sie die Schlüsselwörter DRIVE und NAME immer angeben müssen. Zulässige Laufwerkkennungen sind: DF0:, DF1:, DF2:, DF3:. Als Diskettenname ist jede beliebige Zeichenkette (String) mit bis zu dreißig Zeichen, aus einem oder mehreren Worten bestehend, erlaubt. Bei mehreren Worten muß der gesamte Name zwischen Anführungsstriche gesetzt werden.

> **Achtung!** FORMAT löscht den Inhalt einer bereits beschriebenen Diskette ohne weitere Warnung. Vergewissern Sie sich, daß keine wichtige Datei auf der zu formatierenden Diskette ist.

FORMAT funktioniert bei allen Disketten und Festplatten-Partitionen (eingeschlossen 5,25-Zoll-Disketten), sofern sie mit dem MOUNT-Kommando initialisiert wurden. FORMAT richtet einen Trashcan (Mülleimer) auf der formatierten Disk ein, falls Sie nicht den Zusatz NOICONS angegeben haben. Mittels <Ctrl>-C können Sie den Formatierungsprozeß auch vorzeitig abbrechen.

Erstellen Sie eine Kopie einer ganzen Diskette mit dem Befehl DISKCOPY, braucht die Zieldiskette übrigens vorher nicht formatiert zu werden.

Beispiel(e):

```
FORMAT DRIVE DF0: NAME Arbeitsdiskette
```

formatiert die Diskette in Laufwerk DF0: und gibt ihr den Namen ARBEITSDISKETTE.

Das Kommando INITIALIZE ist der FORMAT entsprechende Workbench-Menüpunkt. Genauso wie FORMAT arbeitet er mit allen Disktypen, die mit MOUNT initialisiert wurden. INITIALIZE kopiert einen Trashcan auf die Diskette. Sollten Sie gerade Ihre SYS:-Diskette in einem Laufwerk haben, dann wird das dafür zuständige Icon von dieser Diskette genommen. Anderenfalls muß die Workbench-Diskette dafür herhalten. Auch INITIALIZE können Sie mit <Ctrl>-C jederzeit abbrechen.

Siehe auch: DISKCOPY, INSTALL, RELABEL

IF

Format: IF[NOT][WARN][ERROR][FAIL][<string>EQ<string>][EXISTS <name>]

Muster: IF "NOT/S,WARN/S,ERROR/S,FAIL/S,EQ/K,EXISTS/K"

Zweck: Abfragen einer Bedingung.

Beschreibung: IF erlaubt Bedingungen in Befehlssequenzen. Dieser Befehl darf nur in einer EXECUTE-Datei verwendet werden. Sind eine oder mehrere Bedingungen erfüllt, führt IF die folgenden Befehle aus, bis er einen Befehl ELSE oder ENDIF findet. Sind andernfalls die Bedingungen nicht erfüllt, führt AmigaDOS alles aus, was hinter dem entsprechenden Befehl ELSE folgt. ENDIF und ELSE sind nur in Befehlssequenzen sinnvoll (siehe auch EXECUTE), die IF enthalten. ENDIF beendet einen IF-Befehl. Wird die IF-Bedingung nicht erfüllt, gibt ELSE den weiteren Ablauf des Befehls an. Es ist zu beachten, daß die Bedingungen und Befehle in einem IF-/ELSE-Befehl mehr als eine Zeile vor ihrem entsprechenden ENDIF umspannen können.

Die folgende Aufstellung zeigt einige Muster zur Anwendung von IF, ELSE und ENDIF:

```
IF <Bedingung>
  Befehl
ENDIF
```

```
IF <Bedingung>
  Befehl
ELSE
  Befehl
ENDIF
```

```
IF <Bedingung>
  Befehl
  IF <Bedingung>
    Befehl
  ENDIF
ENDIF
```

Beachten Sie bitte, daß ELSE nach Bedarf eingesetzt werden kann und daß verschachtelte IFs zum nächstgelegenen ENDIF springen.

Der Parameter ERROR ist nur verfügbar, wenn FAILAT größer 10 gesetzt wurde. In gleicher Weise ist FAIL nur verfügbar, wenn FAILAT größer 20 gesetzt wurde.

Schlüsselwort	Funktion
NOT	kehrt das Ergebnis um
WARN	erfüllt, wenn der letzte Fehlercode größer oder gleich 5 ist.
ERROR	erfüllt, wenn der letzte Fehlercode größer oder gleich 10 ist.
FAIL	erfüllt, wenn der letzte Fehlercode größer oder gleich 20 ist.
<a> EQ <b>	erfüllt, wenn der Text von a und b identisch ist.
EXISTS <Datei>	erfüllt, wenn <Datei> existiert.

IF (Fortsetzung)

IF EQ kann mit folgender Form dazu verwendet werden, einen ungesetzten Parameter in einer Befehlssequenz zu entdecken:

```
IF "<a>" EQ ""
  ECHO "Parameter nicht definiert!"
  SKIP ende
ENDIF
```

Beispiel:

```
IF EXISTS arbeit/file
  TYPE arbeit/file
ELSE
  ECHO "File nicht gefunden!"
ENDIF
```

Gibt es das File FILE im Directory ARBEIT, wird es am Bildschirm ausgegeben. Gibt es das File nicht, erscheint die Fehlermeldung FILE NICHT GEFUNDEN! und der nächste Befehl wird ausgeführt.

```
IF ERROR
  SKIP fehlerlab
ENDIF
```

Verzweigt zum (vorher erstellten) LAB fehlerlab, wenn ein Fehler größer gleich Fehlercode 10 aufgetreten ist.

```
IF ERROR
  IF EXISTS fred
    ECHO "File >fred< gibt es, aber es wurde vorher ein Fehler entdeckt."
  ENDIF
ENDIF
```

Siehe auch: FAILAT, SKIP, LAB, EXECUTE, QUIT

INFO

Format: INFO

Muster: INFO

Zweck: INFO erteilt Informationen über die Dateiverwaltung.

Beschreibung: Dieser Befehl zeigt in einer Bildschirmzeile Informationen über jedes Diskettenlaufwerk an. Dazu gehören die maximale Speicherkapazität der Diskette in Kilobyte, der momentan belegte und der freie Speicherplatz in Kilobyte und die freie und belegte Kapazität in Prozent, die Zahl der Disketten-Fehler, die aufgetreten sind, der Disketten-Status und der Name der Diskette.

Beispiel(e):

```
INFO
```

ergibt folgende Bildschirmausgabe:

```
Unit    Size    Used    Free    Full    Errs    Status  Name
DF1:    880K    5       1753    0%      0       Read/Write      Testdisk
DF0:    880K    1699    59      96%     0       Read/Write      CLIDISK

Volumes available:
Testdisk [mounted]
CLIDISK [mounted]
```

INSTALL

Format: INSTALL[DRIVE]<drive>

Muster: INSTALL"DRIVE/A"

Zweck: Installieren der *Boot-Sektoren* auf einer AmigaDOS-Diskette.

Beschreibung: INSTALL macht eine formatierte Diskette zu einer startfähigen Diskette. Nach INSTALL geben Sie einfach die Bezeichnung des Laufwerks an, in dem sich die Diskette befindet, auf die die *Boot-Sektoren* geschrieben werden sollen. Die vier möglichen Laufwerksbezeichnungen sind: DF0:, DF1:, DF2: und DF3:.

Beispiel(e):

```
INSTALL DF0:
```

macht die Diskette in Laufwerk DF0: zu einer Startdiskette.

JOIN

Format: JOIN <name><name>[<name>*]AS<name>

Muster: JOIN ",,,,,,,,,,,,,AS/A/K"

Zweck: JOIN verkettet bis zu 15 Dateien zu einer neuen Datei.

Beschreibung: AmigaDOS kopiert die bezeichneten Dateien in der angegebenen Reihenfolge hintereinander in die als AS definierte neue Datei. Der Name der neuen Datei darf mit keinem der eingelesenen Dateinamen übereinstimmen. Die eingelesenen Originaldateien bleiben unverändert, während die neuentstandene Datei eine Kopie aller eingelesenen Dateien beinhaltet.

Beispiel(e):

```
JOIN teil1 teil2 AS textfile
```

kopiert die angegebenen Files TEIL1 und TEIL2 in die neue Datei TEXTFILE. Die Originalfiles bleiben bestehen.

LAB

Format: LAB <string>

Muster: LAB <text>

Zweck: Sprungziele definieren, die mit SKIP angesprungen werden.

Beschreibung: LAB fügt Sprungziele in Dateien mit Befehlssequenzen ein. Der Befehl ignoriert alle eingegebenen Parameter. Mit LAB wird ein Sprungziel definiert, das anschließend mit dem Befehl SKIP angesprungen wird.

Beispiel(e):

```
LAB ende
```

definiert das Sprungziel ende, das mit SKIP ende angesprungen wird.

Siehe auch: SKIP, EXECUTE, IF

LIST

Format: LIST [[DIR]<dir>] [P|PAT<muster>] [KEYS] [DATES] [NODATES] [TO<name>] [S<string>] [SINCE<datum>] [UPTO<datum>] [QUICK]

Muster: LIST "DIR, P=PAT/K, KEYS/S, DATES/S, NODATES/S, TO/K, S/K, SINCE/K, UPTO/K, QUICK/S"

Zweck: LIST untersucht und listet bestimmte Informationen über ein Directory oder eine Datei auf.

Beschreibung: Der erste Parameter, den LIST akzeptiert, ist DIR. Er bietet drei Möglichkeiten. Erstens: DIR kann ein Dateiname sein, dann zeigt LIST die Datei-Information für diese eine Datei an. Zweitens kann DIR ein Directory-Name sein. Hier zeigt LIST die Datei-Informationen der Dateien und anderen Directories innerhalb des bezeichneten Directory an. Geben Sie gar keinen Namen für DIR an, zeigt LIST den Inhalt des aktuellen Directory. Weitere Einzelheiten über das aktuelle Directory siehe auch unter dem Befehl CD.

LIST sortiert übrigens nicht – wie der Befehl DIR – das Directory, bevor es auf dem Bildschirm aufgelistet wird.

Der Befehl LIST, ohne weitere Angaben, zeigt folgendes an:

```
filename  Größe Status Datum Zeit
:Kommentar
```

Diese Angaben bedeuten:

filename: Name der Datei oder des Directory.

Größe: Die Länge der Datei in Bytes. Wenn sich nichts in der Datei befindet, wird EMPTY für leer gemeldet. Bei Directories wird DIR angezeigt.

Status: Bezeichnet den verfügbaren Zugriff auf diese Datei. Die Buchstaben »rwed« stehen für Read, Write, Execute und Delete.

Datum/Zeit: Datum und Uhrzeit, zu der diese Datei angelegt wurde.

Kommentar: Der mit FILENOTE angehängte Kommentar. Er beginnt immer mit einem Doppelpunkt (:).

LIST (Fortsetzung)

Die zusätzlichen Möglichkeiten bedeuten:

TO	kennzeichnet eine Datei, in die das Ergebnis von LIST geschrieben werden soll. Falls weggelassen, geht die Ausgabe an das aktuelle CLI-Fenster.
KEYS	gibt die Blocknummer jedes Dateikopfes oder Directory an.
DATES	gibt das Datum in der Form TT-MMM-JJ mit an. Diese Option gehört zur Grundeinstellung, außer es wird zusätzlich QUICK verwendet.
NODATES	unterdrückt die Ausgabe von Datum und Zeit.
SINCE <Datum>	listet nur Dateien auf, die am bezeichneten Datum oder später bearbeitet wurden. Das Datum kann in der Form TT-MMM-JJ oder als Wochentag der vergangenen Woche (zum Beispiel MONDAY) eingegeben werden. Ebenfalls erlaubt ist TODAY oder YESTERDAY (für »heute« oder »gestern«).
UPTO <Datum>	listet nur Dateien auf, die am bezeichneten Datum oder früher bearbeitet wurden. Sonst wie SINCE.
P <Muster>	sucht nach Dateien, deren Namen mit dem angegebenen Kriterium übereinstimmt.
S <Zeichenkette>	sucht nach Dateien, deren Name <Zeichenkette> enthält.
QUICK	zeigt nur die Namen der Dateien und Directories an – wie der Befehl DIR.

Die Anordnung der Dateinamen kann in zwei Arten aufgelistet werden. Die einfachste Art und Weise ist der Gebrauch des Schlüsselwortes S, das die Auflistung auf die Dateien beschränkt, in der die bezeichnete Zeichenkette enthalten ist. Wollen Sie eine kompliziertere Suchformel verwenden, dann benutzen Sie das P- oder PAT-Schlüsselwort. Der Eingabe folgt ein passendes Kriterium, das nun genauer beschrieben wird.

Ein Kriterium enthält eine Anzahl spezieller Zeichen mit eigenen Bedeutungen. Ebenso andere Zeichen, um die oben beschriebenen zu vergleichen.

Die speziellen Zeichen sind: ´ ( ) ? % # und |

LIST (Fortsetzung)

Um die spezielle Wirkung dieser Zeichen aufzuheben, muß ihnen ein Apostroph (´) vorangestellt werden. So löscht ´? das Zeichen ? und ´´ das Zeichen

? — entspricht einem einzelnen Zeichen.
% — entspricht einem Nullstring.
<p> — entspricht null oder mehreren Wiederholungen des Kriteriums <p>.
<p1> <p2> — entspricht der Reihe von Kriterien <p1> gefolgt von <p2>.
<p1>|<p2> — entspricht der Übereinstimmung von Kriterium <p1> oder <p2>.
(und) — gruppieren Kriterien .

Also:

LIST PAT A#BC — listet AC ABC ABBC und so weiter.
LIST PAT A#(B|C)D — listet AD ABD ABCD und so weiter.
LIST PAT A?B — listet AAB ABB ACB und so weiter.
LIST PAT A#?B — listet AB AXXB AZXQB und so weiter.
LIST PAT '?#?'# — listet ?# ?AB# ??## und so weiter.
LIST PAT A(B|%)#C — listet A ABC ACCC und so weiter.
LIST PAT#(AB) — listet AB ABAB ABABAB und so weiter.

Beispiel:

```
LIST
```

zeigt alle Files und Directories im aktuellen Directory auf.

```
LIST arbeit S neu
```

listet alle Files und Directories aus dem Directory ARBEIT, die den String NEU in ihrem Namen haben. Beachten Sie, daß die Eingabe:

```
LIST S
```

die Fehlermeldung ARGS NO GOOD FOR KEY verursacht, auch, wenn es das Directory S im aktuellen Directory gibt. Dessen Inhalt erhalten Sie mit der Eingabe:

LIST "S"

LIST (Fortsetzung)

Die Eingabe von

```
LIST arbeit P neu#?(x|y)
```

listet alle Files, die mit »neu« beginnen und mit »x« oder »y« enden.

```
LIST QUICK TO listfile
```

schreibt den Namen der Files ohne weitere Angaben jeweils in eine neue Zeile des Files LISTFILE. Nun können Sie zum Beispiel das File LISTFILE editieren, indem Sie die erste und letzte Zeile löschen und vor jeden Namen den Befehl TYPE schreiben. Geben Sie dann noch ein:

```
EXECUTE listfile
```

dann werden der Reihe nach alle Files, deren Name in listfile stehen, auf dem Bildschirm ausgegeben.

Siehe auch: DATE, DIR, FILENOTE, PROTECT

MAKEDIR

Format: MAKEDIR <dir>

Muster: MAKEDIR "/A"

Zweck: Errichten eines Directory.

Beschreibung: MAKEDIR erzeugt ein neues Directory mit dem angegebenen Namen unterhalb des aktuellen Directory. Der Befehl erzeugt nur ein Directory im aktuellen Directory, daher müssen alle höheren Directories bereits bestehen.

Beispiel(e):

```
MAKEDIR test
```

erzeugt ein Directory mit Namen TEST im aktuellen Directory.

```
MAKEDIR DF1: xyz
```

erzeugt das Directory XYZ im Hauptdirectory der Diskette in Laufwerk DF1:.

```
MAKEDIR DF1:XYZ/ABC
```

erzeugt das Directory ABC im Directory XYZ auf der Diskette im Laufwerk DF1:. Das Directory XYZ muß dazu bereits bestehen.

Siehe auch: DELETE

NEWCLI

Format: NEWCLI [<fenster>] [FROM <executedatei>]

Muster: NEWCLI "FENSTER, FROM"

Zweck: Ein neues CLI-Fenster öffnen.

Beschreibung: AmigaDOS öffnet einen neuen CLI-Prozeß mit einem neuen Fenster. Das neue Fenster wird automatisch das aktuelle. Es hat das gleiche aktuelle Directory und den gleichen *Prompt* wie das, aus dem heraus es geöffnet wurde.

Wird NEWCLI ohne Argument angegeben, erzeugt AmigaDOS ein Fenster in der üblichen Größe und an gewohnter Stelle. Um die Größe des Fensters zu verändern, bringen Sie den Mauszeiger an die untere rechte Ecke, dem Feld zum Ändern der Fenstergröße, betätigen den linken Mausknopf und halten ihn gedrückt. Darauf können Sie das Fenster mit der Maus vergrößern oder verkleinern. Um die Fensterposition zu ändern, bewegen Sie den Mauszeiger zur Kopfleiste, betätigen wiederum den linken Mausknopf und bringen die Maus an die Stelle, an der das Fenster gewünscht wird.

Um das CLI-Fenster den Benutzerwünschen anzupassen, kann man jedoch die genaue Position und Größe und einen neuen Namen eingeben. Die Syntax für ein neues Fenster generell lautet:

```
CON:x/y/Breite/Höhe/Name
```

wobei CON: für ein Fenster steht, X und Y für die Koordinaten der Fensterposition, Breite und Höhe für die Fenstergröße. Name ist die Zeichenkette, die in der Titelzeile stehen soll. Ein Name muß nicht eingegeben werden, der letzte Schrägstrich (/) ist dagegen immer zu setzen. Alle Größen beziehen sich auf Bildschirmpixel.

Beispiel(e):

```
NEWCLI
```

erzeugt ein neues CLI und macht es zum aktuellen.

```
NEWCLI CON:10/30/300/188/MEINCLI
```

erzeugt ein neues CLI. Das Fenster beginnt bei Position 10,30 und ist 300 * 100 Pixel groß. Es trägt den Namen MEINCLI.

NEWCLI (Fortsetzung)

```
NEWCLI "CON:10/30/300/100/MEIN CLI"
```

erzeugt das gleiche Fenster, jedoch erlauben die Anführungszeichen die Verwendung von Leerzeichen innerhalb des Namens. Nähere Informationen über das Device CON: finden Sie in Abschnitt *1.3.6 Zum Verständnis von Device-Namen* in diesem Handbuch.

Unter AmigaDOS 1.2 existiert ein neues Schlüsselwort für das Kommando NEWCLI: Geben Sie das Schlüsselwort FROM und dahinter den Namen einer EXECUTE-Datei an, so führt das neue CLI sofort nach seiner Einrichtung die darin enthaltenen Befehle aus.

Der Notizblock kann nun auch vom CLI aus gestartet werden. Die dazu notwendige Syntax lautet:

```
[run] notepad [[[-q]<filename>]|?]
```

wobei die Option »-q« den Notizblock öffnet, ohne die Fonts von Disk zu laden, <filename> ist der Name des zu öffnenden Files, und das Fragezeichen zeigt Ihnen die Syntax des Befehls, ohne den Notizblock zu öffnen.

Siehe auch: ENDCLI, RUN

PATH

Format: PATH [SHOW] | [ADD <directoryname> [, <directoryname>]...] | [RESET <directoryname> [, <directoryname>]...]

Zweck: Directories, in denen AmigaDOS nach einem Kommando sucht, werden hinzugefügt, geändert oder ausgegeben.

Beschreibung: Normalerweise sucht AmigaDOS zunächst in dem aktuellen Directory, dann im Verzeichnis SYS:C nach einem auszuführenden Befehl. Geben Sie PATH (mit RESET) ohne Directory-Namen ein, dann wird diese Einstellung vorgenommen. Fügen Sie allerdings einen oder mehrere Directory-Namen an, dann wird der momentane Suchpfad durch diese Verzeichnisse ersetzt. Mit dem Zusatz ADD werden die angegebenen Verzeichnisse lediglich angefügt.

Sie können bis zu zehn Verzeichnisse mit einem einzigen PATH ADD hinzufügen. Die Namen müssen durch mindestens ein Leerzeichen voneinander getrennt sein.

Durch PATH RESET wird der alte durch Ihren neuen Suchpfad ausgetauscht (das momentane Directory und SYS:C bleiben allerdings immer erhalten).

PATH SHOW oder einfach nur PATH bringt eine Liste des momentanen Suchpfades auf den Bildschirm. Die Schlüsselwörter RESET, ADD und SHOW können am Anfang oder am Ende der Parameterzeile angegeben werden.

Beispiel(e):

```
PATH SHOW    (oder PATH)
```

Zeigt die Verzeichnisse an, in denen AmigaDOS Kommandos sucht. Ein typischer Suchpfad lautet:

```
Current directory
Workbench: System
C:
```

Geben Sie danach den folgenden Befehl ein, wird AmigaDOS nun auch im Verzeichnis SYS:newcommands nach Kommandos suchen:

```
PATH ADD SYS:newcommands
PATH
```

```
Current directory
Workbench: System
Workbench: newcommands
C:
```

PATH (Fortsetzung)

Sie können dieses Kommando in Ihre Startup-Sequenz übertragen, so daß es automatisch beim Booten dieser Diskette aufgerufen wird. Schauen Sie doch einmal in Ihre Startup-Sequenz. Der Befehl PATH wird dort sicherlich auftauchen!

Eine beliebte Anwendung dieses Befehls besteht darin, nur einige oft benötigte Kommandos aus dem C-Directory in die RAM-Disk zu kopieren. Wenn Sie aber trotzdem noch auf die anderen Kommandos zugreifen wollen, müssen Sie mit ASSIGN und PATH mitteilen, daß AmigaDOS sowohl im RAM-Disk-C-Directory als auch in dem C-Directory auf der Workbench-Disk nachschauen soll.

PROMPT

Format: PROMPT<prompt>

Muster: PROMPT "PROMPT"

Zweck: Verändern des Prompt.

Beschreibung: Mit PROMPT kann das CLI-Bereitschaftszeichen (*Prompt*) im aktuellen CLI verändert werden. Geben Sie keine Parameter an, setzt AmigaDOS das CLI-Zeichen auf Standard-Prompt »n>« zurück. Anderenfalls wird das bisherige Prompt durch die eingegebene Zeichenkette ersetzt. AmigaDOS akzeptiert auch eine spezielle Zeichenkombination, die Zeichen »%N«. Die Funktion dieser Zeichen wird im 2. Beispiel demonstriert.

Beispiel(e):

```
PROMPT
```

setzt das Prompt zum normalen Zeichen zurück.

```
PROMPT "%N>"
```

setzt das Prompt zum normalen Zeichen zurück. Das »%N« steht für die Nummer des aktuellen CLI-Prozesses.

PROTECT

Format: PROTECT[FILE]<filename>[FLAGS<status>]

Muster: PROTECT"FILE,FLAGS/K"

Zweck: Zugriffe auf eine Datei einschränken.

Beschreibung: Der Befehl PROTECT erlaubt es, die Benutzung einer beliebigen Datei einzuschränken und sie somit zu schützen. Das Schlüsselwort FLAGS läßt vier Möglichkeiten zu: Lesen (r), Schreiben (w), Löschen (d) und Ausführen (e). Um diese Möglichkeiten zu bezeichnen, müssen Sie ein »r«, »w«, »d« oder ein »e« nach dem Dateinamen eingeben. Wenn nach FLAG kein Parameter steht, nimmt PROTECT an, daß keiner der vier Zugriffe auf die Datei erlaubt werden soll. Werden alle Parameter außer »d« gesetzt, kann die Datei nicht gelöscht werden.

Hier noch eine Information für Spezialisten: Unter DOS 1.2 wurde das Kommando PROTECT dahingehend modifiziert, daß es nur noch die unteren vier Bits des Protection-Feldes ändert, die anderen Bits unverändert beläßt. Früher wurden alle Bits auf 1 gesetzt. Das Bit 4 des Feldes (Archiv-Bit) wird nun stets dann gelöscht, sobald ein File geschlossen wird, das beschrieben wurde, oder sobald in ein Directory geschrieben wird. Das ermöglicht einem Archivierungs-Programm, eine Diskette nach solchen Files zu durchsuchen, die seit dem letzten Aufruf des Programms verändert wurden.

Beispiel(e):

```
PROTECT test FLAGS r
```

erlaubt nur Lesezugriffe auf die Datei TEST.

```
PROTECT programm2 rwd
```

erlaubt nur die Zugriffe lesen, schreiben und löschen auf PROGRAMM2.

Ist der Zugriff »schreiben« nicht erlaubt, kann die Datei zum Beispiel nicht mit COPY überschrieben werden, Zugriffe mit dem Editor sind aber weiterhin möglich. Ebenso kann der Inhalt eines Files mit TYPE auch dann noch ausgegeben werden, wenn der Zugriff »lesen« nicht erlaubt ist. Lediglich der folgende Befehl ist nicht möglich:

```
DATE < test? !
```

Siehe auch: LIST

QUIT

Format: QUIT [<returncode>]

Muster: QUIT "RC"

Zweck: Eine Befehlssequenz abbrechen, wenn ein Fehler auftaucht.

Beschreibung: QUIT unterbricht eine Befehlssequenz und gibt dem Benutzer eine Fehlernummer aus. Dieser Befehl ist nur in Befehlssequenzen sinnvoll.

Beispiel:

```
QUIT
```

Der Befehl unterbricht die Befehlssequenz ohne Fehlermeldung.

```
FAILAT 30
IF ERROR
  QUIT 20
ENDIF
```

Ist beim letzten Kommando ein Fehler aufgetaucht, dann unterbricht diese Sequenz mit der Fehlernummer 20. Mehr über Befehlssequenzen finden Sie in der Beschreibung von EXECUTE in diesem Kapitel.

Siehe auch: EXECUTE, IF, LAB, SKIP

RELABEL

Format: RELABEL [DRIVE]<drive> [NAME]<name>

Muster: RELABEL "DRIVE/A, NAME/A"

Zweck: Ändern eines Diskettennamens.

Beschreibung: RELABEL ändert den Namen einer Diskette in den hinter NAME bezeichneten neuen Namen. Diskettennamen werden üblicherweise beim Formatieren einer Diskette festgesetzt.

Beispiel:

```
RELABEL DF1: "Neuer Name"
```

ändert den Namen der Diskette im Laufwerk DF1: zu NEUER NAME.

Siehe auch: FORMAT

RENAME

Format: RENAME [FROM]<name> [TO|AS]<name>

Muster: RENAME "FROM/A, TO=AS/A"

Zweck: Ein File oder Directory unbenennen oder Files beziehungsweise Directories innerhalb des File-Systems bewegen.

Beschreibung: RENAME ändert den Namen des nach FROM angegebenen Files oder Directory zu dem nach TO eingegebenen. Die Datei hinter FROM muß auf der aktuellen Diskette stehen. Benennen Sie mit RENAME ein Directory um, wird dessen Inhalt natürlich nicht verändert. Mit RENAME kann aber auch die Position eines Directory oder Files innerhalb der File-Hierarchie verändert werden. Zum Beispiel: RENAME :BERND/BRIEF TO :MARIA/BRIEF schreibt die Datei BRIEF aus dem Directory BERND in das Directory MARIA, ohne dessen Namen dabei zu ändern. Auch mit ganzen Directories ist das möglich. Andere Betriebssysteme nennen diesen Vorgang *moving*, also *bewegen*. Bildlich gesprochen kann auch gesagt werden: Mit RENAME kann ein File von einer Schublade der Diskette in eine andere gelegt werden. Das Ganze ist natürlich auch mit COPY und DELETE möglich. Bei RENAME wird aber die Datei nicht tatsächlich kopiert, sondern ihr Name nur anders eingeordnet. Das ist natürlich ein erheblicher Unterschied.

RENAME wird nicht ausgeführt, wenn das TO-File bereits existiert. Damit soll ein versehentliches Löschen von Files vermieden werden.

Beispiel(e):

```
RENAME arbeit/test AS :test/testfile
```

schreibt das File TEST aus dem Directory ARBEIT als File TESTFILE in das Directory TEST. Das Directory TEST muß im Hauptdirectory der Diskette bestehen.

RUN

Format: RUN <Befehlsliste>

Muster: RUN Befehl+
Befehl.....

Zweck: RUN startet Tasks (Programme), die im Hintergrund abgearbeitet werden.

Beschreibung: RUN erzeugt einen CLI-Prozeß, der nicht im interaktiven Dialog abläuft. Der restliche Teil der Befehlszeile gilt als Eingabe für diesen Prozeß. Das Hintergrund-CLI führt die Befehle nacheinander aus und beendet sich dann selbst. RUN zeigt die laufende Nummer des erzeugten Prozesses an.

Das neue CLI hat das gleiche aktuelle Directory und denselben Befehls-Stapelspeicher wie das CLI, in der RUN gestartet wurde.

Mehrere Befehle, die im Hintergrund abgearbeitet werden sollen, können Sie durch ein Plus-Zeichen »+« und anschließende Betätigung der RETURN-Taste verbinden. RUN versteht die nächste Zeile, nach einem »+RETURN« als Fortsetzung der gleichen Zeile. Auf diese Weise ist es möglich, eine einzelne Befehlszeile aus mehreren Zeilen mit je einem Plus-Zeichen am Zeilenende zusammenzustellen.

Beispiel:

```
RUN COPY test TO PRT:+
DELETE test+
ECHO "Datei gelöscht"
```

Diese Sequenz startet ein Hintergrund-Task, das das File TEST ausdruckt, dann löscht und zum Schluß die Meldung DATEI GELÖSCHT auf den Bildschirm ausgibt.

```
RUN EXECUTE Befehle
```

führt das EXECUTE-File BEFEHLE in einem Hintergrund-Task aus.

SEARCH

Format: SEARCH [FROM]<name>|<muster> [SEARCH]<string> [ALL]

Muster: SEARCH "FROM, SEARCH/A,ALL/S"

Zweck: Suchen eines String innerhalb angegebener Dateien.

Beschreibung: SEARCH sucht nach einer bezeichneten Zeichenkette in allen Dateien eines Directory. SEARCH durchsucht auch alle Unterdirectories, wenn Sie den Zusatz ALL anhängen. SEARCH gibt alle Zeilen aus, die die angegebene Zeichenkette beinhalten. Zudem wird der Name der Datei angezeigt, in der gerade gesucht wird.

Der Directory-Name kann, wie unter LIST beschrieben, durch Joker eingegrenzt werden. Eine vollständige Beschreibung der Joker und sonstigen Kriterien findet sich beim Befehl LIST in diesem Kapitel. Verwenden Sie einen Joker, durchsucht SEARCH nur solche Dateien, die den bezeichneten Kriterien entsprechen. Der Directory-Name kann also ein bezeichnetes Directory oder ein Joker sein.

AmigaDOS sucht die Zeichenkette entweder in Groß- oder Kleinschrift. Wenn die angegebene Zeichenkette Leerzeichen enthält, muß der Text in Anführungsstriche gesetzt werden.

Falls notwendig, kann der Befehl mit der Tastenkombination <Ctrl>-C abgebrochen werden. Um die Suche in der gerade durchsuchten Datei abzubrechen und in der nächsten Datei fortzusetzen, sollten Sie <Ctrl>-D betätigen.

Beispiel(e):

```
SEARCH arbeit SEARCH test
```

Durchsucht im Directory ARBEIT alle Files nach dem String »test«.

```
SEARCH DF1: SEARCH test ALL
```

Durchsucht alle Dateien und alle Dateien in Unter-Directories nach der Zeichenkette »test«.

SETCLOCK

Format: SETCLOCK OPT LOAD | SAVE

Muster: "SETCLOCK OPT/A/K"

Zweck: Übergabe der Systemzeit an die Hardware-Uhr oder Übernahme der Systemzeit von der Hardware in das Betriebssystem.

Beschreibung: Falls Sie glücklicher Besitzer eines Amiga 2000 oder eines Amiga 500 mit Speichererweiterung sind, verfügt Ihr Rechner über eine batteriegepufferte Hardware-Uhr zusätzlich zur systeminternen altbekannten Uhr des Amiga. Wollen Sie diese Hardware-Uhr nun stellen, dann sollten Sie mit dem Programm PREFERENCES auf der Workbench-Diskette oder mit dem CLI-Kommando DATE zunächst einmal die Systemuhr richtig einstellen. Geben Sie im CLI ein:

SETCLOCK OPT SAVE

Nun wird die Systemzeit in die Hardware-Uhr übertragen, und selbst ein Ausschalten des Rechners kann die Zeit dort nicht mehr vertreiben.

Bei RESET oder beim Einschalten des Rechners muß nun allerdings ihrerseits die Hardware-Zeit in die zu diesem Zeitpunkt natürlich nicht richtig eingestellte Systemuhr übertragen werden. Das geschieht mit dem Kommando:

SETCLOCK OPT LOAD

Es empfiehlt sich, dieses Kommando in die Startup-Sequenz zu übernehmen, so daß die Hardware-Zeit bei jedem Boot-Vorgang auch in die Systemuhr übernommen wird.

Je nach der Version der Workbench-Diskette, die Sie zusammen mit Ihrem Computer bekommen haben, kann der entsprechende Befehl zum Setzen der Zeit auch SETTIME heißen.

Siehe auch: SETTIME

SETDATE

Format: SETDATE <file> <date> [<time>]

Zweck: Ändern des Zeiteintrages für ein File/Directory.

Beschreibung: Mit diesem Kommando können Sie Datums- und Zeiteintrag eines Files oder Directory verändern. Geben Sie dabei das Datum in der folgenden Form tt-mmm-jj (oder den Wochentag oder yesterday und so weiter) an.

Die Zeit sollte das Format ss:mm:ss (oder: ss:mm) haben.

SETMAP

Format: SETMAP <mapfilename>

Zweck: Ändert die landesabhängige Tastaturbelegung.

Beschreibung: Es existieren mehrere alternative Tastaturbelegungen, die in den verschiedenen Ländern verwendet werden. Das Directory :DEVS/KEYMAPS enthält die Namen der verfügbaren Tastaturbelegungen. Um auf die im ROM gespeicherte Belegung umzuschalten, geben Sie bitte ein:

```
SETMAP usa
```

Sie können dieses Kommando selbstverständlich in Ihre Startup-Sequenz übernehmen.

SETTIME

Format: SETTIME [-e] | [-i]

Zweck: Übergabe der Systemzeit an die Hardware-Uhr oder Übernahme der Systemzeit von der Hardware in das Betriebssystem.

Beschreibung: Falls Sie glücklicher Besitzer eines Amiga 2000 oder eines Amiga 500 mit Speichererweiterung sind, verfügt Ihr Rechner über eine batteriegepufferte Hardware-Uhr zusätzlich zur systeminternen altbekannten Uhr des Amiga. Wollen Sie diese Hardware-Uhr nun stellen, dann sollten Sie mit dem Programm PREFERENCES auf der Workbench-Diskette oder mit dem CLI-Kommando DATE zunächst einmal die Systemuhr richtig einstellen. Geben Sie im CLI ein:

SETTIME -i

Nun wird die Systemzeit in die Hardware-Uhr übertragen, und selbst ein Ausschalten des Rechners kann die Zeit dort nicht mehr vertreiben.

Bei RESET oder beim Einschalten des Rechners muß nun allerdings ihrerseits die Hardware-Zeit in die zu diesem Zeitpunkt natürlich nicht richtig eingestellte Systemuhr übertragen werden. Das geschieht mit dem Kommando:

SETTIME -e

Es empfiehlt sich, dieses Kommando in die Startup-Sequenz zu übernehmen, so daß die Hardware-Zeit bei jedem Boot-Vorgang auch in die Systemuhr übernommen wird.

Je nach der Version der Workbench-Diskette, die Sie zusammen mit Ihrem Computer bekommen haben, kann der entsprechende Befehl zum Setzen der Zeit aber auch SETCLOCK heißen.

Siehe auch: SETCLOCK

SKIP

Format: SKIP <label>

Muster: SKIP "LABEL"

Zweck: Sprung zu einem definierten Label in einer Befehls-Sequenz.

Beschreibung: SKIP führt einen Sprung in einer EXECUTE-Befehls-Sequenz aus. Der Befehl SKIP wird in Verbindung mit LAB verwendet, ohne daß dabei Befehle ausgeführt werden.

SKIP können Sie mit oder ohne Label verwenden. Ohne Label-Eingabe findet SKIP den nächsten unbenannten LAB-Befehl. Mit der Angabe eines Labels versucht AmigaDOS, einen LAB-Befehl mit dem gleichen Labelnamen zu finden. LAB muß das erste Wort in einer Zeile der Datei sein. Wenn SKIP das bezeichnete Label nicht finden kann, wird die Befehlsfolge beendet, und AmigaDOS gibt folgende Meldung aus:

```
label »name« not found by Skip
```

SKIP springt in einer Befehls-Sequenz nur nach vorne.

Beispiel(e):

```
SKIP
```

springt zum nächsten Label, welches keinen Namen trägt.

```
IF ERROR
 SKIP fehlerbearbeitung
ENDIF
```

springt zum Label »fehlerbearbeitung«, wenn in den vorhergehenden Befehlen ein Fehler aufgetreten ist.

SKIP (Fortsetzung)

Eine solche oder ähnliche Sequenz kann zum Beispiel wie im folgenden Beispiel verwendet werden:

```
FAILAT 100
ASSEM text
IF ERROR
  SKIP fehler
ENDIF
LINK
SKIP fertig
LAB fehler
ECHO "Fehler bei ASSEM"
LAB fertig
ECHO "Nächster Befehl bitte"
```

Siehe auch: EXECUTE, LAB, IF, FAILAT, QUIT

SORT

Format: SORT [FROM]<name> [[TO]<name>] [COLSTART<n>]

Muster: SORT "FROM/A, TO/A, COLSTART/K"

Zweck: Textdateien zeilenweise sortieren.

Beschreibung: Mit SORT lassen sich Dateien sortieren. Dieser Befehl ist allerdings für lange Dateien nicht schnell genug und er kann keine Dateien sortieren, die nicht in den Speicher passen.

Sie bezeichnen das zu sortierende File als FROM-Datei, das Ergebnis geht in die Datei, die Sie hinter TO angeben. SORT geht davon aus, daß die FROM-Datei eine normale Textdatei ist, in der jede Zeile mit einem Wagenrücklauf (carriage return) beendet wurde. Die Datei wird zeilenweise alphabetisch sortiert, ohne Rücksicht auf Groß- oder Kleinschreibung.

Um dies in beschränktem Maß zu umgehen, geben Sie das Schlüsselwort COLSTART ein, um diejenige Spalte festzulegen, ab der der Vergleich stattfinden soll. SORT vergleicht dann die Buchstaben der Zeile ab dem bezeichneten Startpunkt bis zum Zeilenende. Lassen sich die Zeilen so nicht eindeutig ordnen, werden die Zeichen vor der mit COLSTART bezeichneten Spalte mit einbezogen.

> **Achtung!** Der normalerweise eingerichtete Stapelspeicher ist nur 4000 Byte groß. Das reicht höchstens zum Sortieren von Files mit 50 Zeilen. Wollen Sie größere Files sortieren, müssen Sie den Stack mit dem Befehl STACK vergrößern. Wieviel Stack Sie brauchen, hängt von der Größe des Files ab und kann nur durch Schätzen ermittelt werden. Haben Sie den Stack aber zu klein gemacht, verweigert der Amiga jede weitere Zusammenarbeit (Absturz) bis zum Neustart mit <Ctrl>-<A>-<A>.

Beispiel(e):

```
SORT test TO sorttest
```

sortiert die Datei TEST zeilenweise und schreibt das Ergebnis in die Datei SORTTEST.

```
SORT index TO sortierter.index COLSTART 4
```

sortiert die Datei INDEX ab der vierten Spalte und schreibt das Ergebnis in die Datei SORTIERTER.INDEX.

Siehe auch: ><, STACK

STACK

Format: STACK [<n>]

Muster: STACK "GRÖSSE"

Zweck: Verändern der Stack-Größe.

Beschreibung: STACK zeigt oder legt die Größe des Stapelspeichers fest. Wenn ein Programm abläuft, benötigt es einen gewissen Speicherplatz zum Zwischenspeichern von Werten und um programminterne Sprünge durchführen zu können. In den meisten Fällen reicht der eingestellte Speicherplatz aus, trotzdem kann er mit dem Befehl STACK verändert werden. Dies erreicht man durch den Befehl STACK, gefolgt von der neuen Stapelspeichergröße in Byte. Der Befehl STACK ohne Argument zeigt die aktuelle Größe des Stapel-Speichers an.

SORT ist normalerweise der einzige Befehl, für den man den Stackwert verändern muß. Rekursive Befehle wie zum Beispiel DIR benötigen allerdings ebenfalls einen erweiterten Stack, wenn mehr als sechs ineinander verschachtelte Directories auftreten.

> **Achtung!** Zu wenig Stack bringt den Amiga außer Tritt. Da hilft nur noch der Neustart. Machen Sie den Stack lieber zu groß als zu klein!

Beispiel(e):

```
STACK 8000
```

setzt die Stack-Größe auf 8000 Byte.

```
STACK
```

zeigt die Größe des aktuellen Stacks an.

STATUS

Format: STATUS [<prozess>] [FULL] [TCB] [SEGS] [CLI|ALL]

Muster: STATUS "PROZESS, FULL/S, TCB/S, SEGS/S, CLI=ALL/S"

Zweck: Informationen über CLI-Prozesse ausgeben.

Beschreibung: Der Befehl gibt Informationen über die momentan bestehenden CLI-Prozesse aus. STATUS alleine listet die Nummern der CLI-Prozesse und die darin arbeitenden Programme auf.

<prozess> bezeichnet den jeweiligen CLI-Prozeß und gibt nur Informationen über diesen einen Prozeß aus. Sonst werden Informationen über alle aktiven CLI-Prozesse ausgegeben.

Das Schlüsselwort FULL führt alle Funktionen der unten erklärten Schlüsselwörter SEGS, TCB und ALL kombiniert aus.

Das Schlüsselwort SEGS zeigt die Namen der Segmente in der Segmentliste eines jeden Prozesses.

TCB zeigt Informationen über Priorität, Stack-Größe und globale Vektorgröße eines jeden Prozesses. Weitere Details zu Stack und globaler Vektorgröße finden sich im *technischen AmigaDOS-Handbuch.*

CLI: Das Schlüsselwort CLI listet CLI-Prozesse und zeigt die Namen der gerade laufenden Befehle, falls vorhanden.

Beispiel(e):

```
STATUS
```

gibt Informationen über alle Prozesse.

```
STATUS 4 FULL
```

gibt alle Informationen über den CLI-Prozeß 4.

TYPE

Format: TYPE [FROM]<name> [[TO]<name>] [OPT N|H]

Muster: TYPE "FROM/A, TO, OPT/K"

Zweck: Ausgabe einer Datei.

Beschreibung: Der Befehl TYPE gibt eine Textdatei oder ein Programm aus. Dabei werden Tabulatoren, die Sie in der Datei gesetzt haben, bei der Ausgabe ausgedehnt. TO bezeichnet die Zieldatei. Lassen Sie diese fort, geht die Ausgabe in den aktuellen Ausgabestrom, das heißt in den meisten Fällen in das aktuelle Fenster.

Die Ausgabe selbst brechen Sie durch Drücken von <Ctrl>-C ab und halten sie durch Drücken der Space-Taste oder irgendeiner anderen Taste an. Betätigen Sie die RETURN-Taste oder <Ctrl>-X, wird die Ausgabe danach wieder aufgenommen.

Mit OPT sind zusätzliche Modifikationen der Ausgabe möglich. Die erste Option »N« sorgt dafür, daß Zeilennummern mit ausgegeben werden, die Option »H« bewirkt die Ausgabe aller Zeichen als ASCII-Code in Hexadezimalzahlen und Klartext.

Beispiel(e):

```
TYPE arbeit/text
```

schreibt den Inhalt der Datei TEXT auf den Bildschirm.

```
TYPE arbeit/text OPT n
```

schreibt den Inhalt von TEXT mit Zeilennummern auf den Bildschirm.

```
TYPE programm TO hexprogramm OPT h
```

schreibt das File PROGRAMM in hexadezimaler Form in die Datei HEXPROGRAMM.

WAIT

Format: WAIT <n> [SEC|SECS][MIN|MINS] [UNTIL <zeit>]

Muster: WAIT "SEC=SECS/S, MIN=MINS/S, UNTIL/K"

Zweck: WAIT wartet eine bestimmte Zeit.

Beschreibung: Der Befehl WAIT verzögert eine Befehlssequenz oder einen mit RUN gestarteten Prozeß für eine gewisse Zeitspanne oder bis zu einem bezeichneten Zeitpunkt. Falls Sie nichts anderes eingegeben haben, wartet der Amiga eine Sekunde lang.

Der eingegebene Parameter ist eine Zahl, die der Wartezeit in Sekunden (oder Minuten, falls das Schlüsselwort MINS eingegeben wird) entspricht.

Verwenden Sie das Schlüsselwort UNTIL, wartet das System bis zu einem gewissen Zeitpunkt, der in der Form SS:MM (Stunde:Minute) eingegeben wird.

Beispiel(e):

```
WAIT
```

hält die Ausführung für eine Sekunde an.

```
WAIT 10 MINS
```

hält die Ausführung 10 Minuten an.

```
WAIT UNTIL 21:15
```

wartet mit der weiteren Ausführung der Befehle, bis die Systemuhr 21.15 Uhr zeigt.

WHY

Format: WHY

Muster: WHY

Zweck: Ausgabe weiterer Informationen über einen aufgetretenen Fehler.

Beschreibung: Tritt bei der Bearbeitung eines Befehls ein Fehler auf, bricht der Amiga die Ausführung des Befehls ab und gibt eine Fehlermeldung aus. Diese enthält normalerweise den Namen des Files (falls ein File das Problem war), zeigt aber keine weiteren Informationen an. Konnte zum Beispiel der Befehl:

```
COPY fred TO *
```

nicht ausgeführt werden, erscheint die Meldung:

```
Can´t open fred
```

Warum FRED nicht geöffnet werden konnte, erfährt der Anwender nicht. So könnte FRED zum Beispiel ein Directory sein, oder der Speicher des Amiga beziehungsweise die Diskette könnten voll sein. AmigaDOS macht keinen Unterschied, der Befehl kann nicht ausgeführt werden. Und der Anwender hat zu wissen, was er falsch gemacht hat. Oder er gibt, bevor er einen neuen Befehl eintippt, das Wörtchen WHY (warum) ein. Nun sagt der Rechner, was nicht stimmte.

Beispiel(e):

```
TYPE DF0:
Can´t open DF0:
WHY
Last command failed because objekt is not of required type
```

»Der letzte Befehl konnte nicht ausgeführt werden, weil das Objekt nicht vom benötigten Typ ist.« Dieser Befehl konnte also nicht ausgeführt werden, weil ein Device nicht auf dem Bildschirm ausgegeben werden kann. WHY gibt also auch nur Hinweise, keine Lösungen aus.

2.2 AmigaDOS-Befehle für Programmierer

ALINK

Format: ALINK [[FROM|ROOT]<filename> [,<filename>*|+<filename>*]]
[TO<name>] [WITH<name>] [LIBRARY|LIB<name>] [MAP <map>]
[XREF <name>] [WIDTH<n>]

Muster: ALINK "FROM=ROOT, TO/K, WITH/K, VER/K, LIBRARY=LIB/K, MAP/K, XREF/K, WIDTH/K"

Zweck: Kompilierte oder assemblierte Object-Files zusammenbinden.

Beschreibung: ALINK weist AmigaDOS an, Object-Files zusammenzubinden. Ebenso werden mit ALINK System-Routinen in ein File eingebunden und Overlay-Files gebildet. Das Output-File von ALINK wird mit dem Systemloader geladen und läuft unter Overlay-Überwachung, sofern nötig. Genauere Informationen über ALINK finden Sie in Kapitel 8, *AmigaDOS-Programmierer-Handbuch.*

Beispiel(e):

```
ALINK a+b+c TO ausgabefile
```

bindet die Assemblerfiles a, b und c zum File ausgabefile.

ASSEM

Format: ASSEM [PROG|FROM]<programm> [-O<code>] [-V<ver>] [-L<listing>] [-E][-H] [-C|OPT<opt>] [-I<dirlist>]

Muster: ASSEM "PROG=FROM/A, -O/K, -V/K, -L/K, -H/K, -E/K, -C=OPT/K, -I/K"

Zweck: Ein Assembler-Programm in Maschinensprache übersetzen.

Beschreibung: Der Befehl assembliert ein Programm in der Maschinensprache des MC 68000. Weitere Einzelheiten finden sich in Kapitel 7, *AmigaDOS-Programmierer-Handbuch.*

PROG ist die Quelldatei.

-O ist die Objektdatei mit der binären Ausgabe des Assemblers.

-V ist die Datei für Meldungen, wird sie nicht genauer bezeichnet, gehen ihre Meldungen in das CLI-Fenster.

-L ist die Listingdatei.

-C bezeichnet dem Assembler die Optionen.

-H ist eine Kopfdatei, die eingelesen wird, als stände sie am Beginn des Quellfiles (ähnlich INCLUDE im Quellfile).

-I stellt eine Liste der Directories auf, die nach enthaltenen Dateien durchsucht werden.

-E ist die Datei, in die die EQU-Anweisungen des Quellcodes geschrieben werden. -E wird zur Erstellung eines Headers gebraucht, das alle diese EQU-Anweisungen enthält.

Die folgenden Optionen können mit OPT oder -C angegeben werden:

S erzeugt eine Symboltabelle als Teil der Objektdatei.

X erzeugt eine Cross-Reference-Datei.

W (Größe) setzt den Arbeitsspeicher auf »Größe« Bytes fest.

Beispiel:

```
ASSEM prog.asm TO prog.obj
```

wandelt den Quellcode in File PROG.ASM in das Objektfile PROG.OBJ um. Alle Fehlermeldungen werden am Terminal ausgegeben, aber es wird kein Assembler-Listing erstellt.

ASSEM (Fortsetzung)

```
ASSEM prog.asm TO prog.obj -h slib -l prog.list
```

assembliert das gleiche Source-File zum gleichen Objekt-File, bindet das File SLIB ein und schreibt das Assembler-Listing in das File PROG.LIST.

```
ASSEM mini.asm -O mini.obj w8000
```

assembliert ein sehr kleines Programm.

DOWNLOAD

Format: DOWNLOAD [FROM]<name> [TO]<name>

Muster: DOWNLOAD "FROM/A, TO/A"

Zweck: Programme von anderen Rechnern einlesen.

Beschreibung: Der Befehl DOWNLOAD dient dem Empfang von Programmen, die auf anderen Rechnern, wie zum Beispiel einer SUN-Workstation, erstellt wurden. DOWNLOAD kann nur in Verbindung mit einem BillBoard (Zusatzgerät für die SUN) verwendet werden. Geben Sie zunächst folgenden Befehl auf der SUN ein:

```
binload -p&
```

(Beachten Sie, daß der Befehl BINLOAD kein Befehl des CLI ist. BINLOAD wird nur auf der SUN verwendet.) Geben Sie dann auf dem Amiga den folgenden Befehl ein:

```
DOWNLOAD sun-Dateiname amiga-Dateiname
```

Bevor Sie die SUN starten, sollten das BillBoard und der Amiga angeschlossen und angeschaltet sein, denn sie werden sonst von der SUN nicht erkannt. Ferner sollte der SUN-Dateiname mit dem Zusatz ».ld« enden und ein *gelinktes Loadfile* bezeichnen.

Wichtig ist, daß DOWNLOAD-Files immer relativ zu dem SUN-Directory verwaltet werden, in dem BINLOAD gestartet wurde. Wenn man sich nicht mehr an das Directory erinnern kann, in dem BINLOAD gestartet wurde, dann muß die volle Directory-Bezeichnung angegeben werden. Sie können den BINLOAD-Befehl auf der SUN mit PS und einem anschließenden KILL auf seinem PID abbrechen. Der Soft-Reset des Computers weist BINLOAD an, eine Meldung in seine übliche Ausgaberichtung zu schreiben. Die voreingestellte Ausgaberichtung ist das Fenster, in dem BINLOAD gestartet wurde. Wenn sich DOWNLOAD »aufhängt«, können Sie den Befehl mit <Ctrl>-C abbrechen.

Kapitel 5, *AmigaDOS-Programmierer-Handbuch* beschreibt im Detail, wie Programme vom IBM PC und von der SUN eingelesen werden, und gibt Hinweise darüber, wie Programme von nicht eigens unterstützten Computern übernommen werden können.

Beispiel(e):

```
binload -p&                     (Aauf der SUN)
DOWNLOAD test.ld test      (auf dem Amiga)
```

oder

```
DOWNLOAD /usr/fred/DOS/test.ld test
```

Diese Befehle laden die angegebenen Files von der SUN zum Amiga.

READ

Format: READ [TO]<name> [SERIAL]

Muster: READ "TO/A, SERIAL/S"

Zweck: READ liest Daten aus der parallelen oder der seriellen Schnittstelle und speichert sie in einer Datei.

Beschreibung: Der Befehl READ erwartet und übernimmt einen Strom von hexadezimalen Zeichen von der parallelen Schnittstelle. Geben Sie den Zusatz SERIAL ein, überwacht READ statt dessen die serielle Schnittstelle. Jedes Hex-Paar wird als ein Byte im Speicher abgelegt. READ behandelt »Q« als Ende des Hex-Stroms. READ erkennt ansonsten nur die ASCII-Zahlen 0 bis 9 und die Großbuchstaben A bis F. READ übergeht Leerzeichen, neue Zeilen und Tabulatoren. Für jedes Nibble der zu empfangenden Datei muß eine ASCII-Hex-Zahl gesendet werden, und sie müssen in gerader Anzahl ankommen. Ist der Datenfluß beendet, schreibt READ die Bytes aus dem Speicher in die bezeichnete Datei. (Sie können sowohl Text- als auch binäre Files übertragen.)

> **Achtung!** Vorsicht, wenn Sie denselben Filenamen zweimal verwenden. READ überschreibt vorhandene Dateien ohne Warnung.

> **Achtung!** Wählen Sie die Baud-Rate nicht zu hoch. Die Gefahr des Datenverlustes ist zu groß.

Beispiel(e):

```
READ TO DF0:neu
```

liest Daten vom parallelen Port und schreibt sie in das File NEU auf die Diskette in Laufwerk DF1:.

```
READ neu SERIAL
```

liest Daten vom seriellen Port und schreibt sie in die Datei NEU.

2.3 Alphabetische Kurzübersicht über die AmigaDOS-Befehle

Anwender-Befehle

File-Behandlung:

;	Kommentarzeichen.
<	Ändert die Eingaberichtung.
>	Ändert die Ausgaberichtung.
ADDBUFFERS	Verkürzung der Disk-Zugriffszeit durch Hinzufügen von Pufferspeicher.
BINDDRIVERS	Einbinden der Device-Driver für zusätzliche Device-Hardware.
CHANGETASKPRI	Änderung der CLI-Task-Priorität.
COPY	Kopieren von Files oder Files in Directories.
DELETE	Löschen von bis zu 10 Dateien oder Directories.
DIR	Anzeigen der Filenamen eines Directory.
DISKCHANGE	Teilt AmigaDOS mit, daß eine 5,25-Zoll Diskette gewechselt wurde.
DISCDOCTOR	Wiederherstellen eventuell zerstörter Disketten.
ED	Aufruf des AmigaDOS-Bildschirmeditors.
EDIT	Aufruf des AmigaDOS-Zeileneditors.
FILENOTE	Hinzufügen einer maximal 80 Zeichen langen Information zu einer Datei
JOIN	Verkettung von maximal 10 Dateien zu einer neuen.
LIST	Ausgabe detaillierter File-Informationen.
MAKEDIR	Erzeugen eines neuen Directory.
PROTECT	Ändern des File-Schutzstatus.
RENAME	Ändern eines Datei- oder Directory-Namens beziehungsweise seiner Position in der Filestruktur.
SEARCH	Durchsuchen einer Reihe von Dateien nach einer Zeichenkette.

SETDATE	Ändern des Zeiteintrages eines File/Directory.
SETMAP	Ändern der Tastaturbelegung.
SORT	Sortieren einfacher Text-Dateien.
TYPE	Ausgabe einer Datei in ASCII- oder HEX-Form.

CLI-Verwaltung:

BREAK	Setzen des Attention-Flags in einem Prozeß (Task).
CD	Festlegen eines aktuellen Directory/Laufwerks.
ENDCLI	Beenden des aktuellen CLI-Prozesses.
NEWCLI	Öffnen eines neuen CLI-Fensters.
PROMPT	Ändern des CLI-Bereitschaftszeichens.
RUN	Ausführen von Befehlen/Programmen als Hintergrundprozesse.
STACK	Zeigen oder ändern der Stackgröße für Kommandos.
STATUS	Informationen über momentan aktive CLI-Prozesse.
WHY	Nähere Erläuterungen zum letzten Fehler.
ECHO	Ausgabe eines ASCII-Strings.
EXECUTE	Ausführen eines Kommando-Files.
FAILAT	Abbruch einer Befehlssequenz bei Fehlern in Abhängigkeit vom Argument.
IF	Bedingte Verzweigung innerhalb einer Befehlssequenz.
LAB	Definition eines Labels (Sprungmarke, siehe SKIP).
QUIT	Beenden einer Befehlssequenz mit Fehler-Code.
SKIP	Sprung zu einer dahinterstehenden Sprungmarke (siehe auch LAB).
WAIT	Warten – eine gewisse Zeit lang oder bis zu einem bestimmten Zeitpunkt.

System- und Speicherverwaltung:

ASSIGN	Zuweisung eines logischen Device-Namens zu einem Directory.
DATE	Ausgabe oder Änderung der Systemzeit.

DISKCOPY	Kopieren einer ganzen Disk mit Formatierung.
FAULT	Ausgabe der zu einem Fehlercode gehörigen Fehlermeldung.
FORMAT	Formatierung einer Diskette oder Festplatte.
INFO	Ausgabe von Informationen über das Dateisystem.
INSTALL	Umformen einer formatierten zu einer startfähigen Diskette.
RELABEL	Änderung des Diskettennamens.
SETTIME / SETCLOCK	Übergabe der Uhrzeit zwischen AmigaDOS und der Hardware-Uhr des Amiga 2000 und Amiga 500.

Befehle für Programmierer

ALINK	Linken von Object-Files.
ASSEM	Assemblieren eines 68000-Assemblerfiles.
DOWNLOAD	Übertragen von Programmen anderer Computer (SUN) in den Amiga.
READ	Einlesen von Daten über die parallele beziehungsweise serielle Schnittstelle.

Kapitel 3: ED – Der Bildschirm-Editor

Dieses Kapitel beschreibt den Bildschirm-Editor ED. Mit ihm können Sie Textfiles erstellen und verändern.

3.1 Einführung in den Bildschirm-Editor

Der Editor kann mit dem Befehl ED gestartet werden. Dabei kann entweder eine neue Datei angelegt oder eine vorhandene bearbeitet werden. Der Text wird auf dem Bildschirm angezeigt und kann vertikal oder horizontal gescrollt werden.

ED erkennt folgendes Muster an:

```
ED "FROM/A, SIZE/K"
```

Um ED aufzurufen, kann man zum Beispiel folgendes eingeben:

```
ED manfred
```

ED versucht, die mit MANFRED bezeichnete Datei zu öffnen. MANFRED ist damit die FROM-Datei. Kann sie geöffnet werden, liest ED die Datei in den Speicher und zeigt die ersten Zeilen auf dem Bildschirm an. Anderenfalls liefert ED ein *leeres Blatt*, bereit, einen neuen Text aufzunehmen. Um den Textspeicher zu vergrößern, so daß er größere Dateien aufnehmen kann, geben Sie den passenden Wert nach dem Schlüsselwort SIZE ein, zum Beispiel so:

```
ED manfred SIZE 45000
```

Die ursprüngliche Größe richtet sich nach der Größe der Datei, ist aber mindestens 40.000 Byte groß.

Es kann übrigens nicht jede Dateiart mit ED eingelesen werden. So akzeptiert ED keine Quelldateien mit Binärcode. Solche Dateien können nur mit dem Zeilen-Editor EDIT überarbeitet werden.

Achtung! Endet eine Datei nicht mit einem Zeilenvorschub, fügt ED diesen an.

Während ED läuft, wird die unterste Zeile zur Ausgabe von Meldungen des Systems und als Befehlszeile benützt. Fehlermeldungen bleiben hier stehen, bis Sie einen neuen Befehl eingeben.

ED-Befehle werden in zwei Kategorien unterteilt:

- unmittelbare Befehle
- erweiterte Befehle

Man verwendet unmittelbare Befehle im Direkt-Modus; entsprechend werden erweiterte Befehle im erweiterten Modus verwendet. ED befindet sich zu Beginn des Editierens stets im Direkt-Modus. Um in den erweiterten Modus zu gelangen, betätigen Sie die Esc-Taste. Nachdem ED die danach eingegebenen Befehle ausgeführt hat, kehrt es automatisch in den Direkt-Modus zurück.

Im Direkt-Modus führt ED die Befehle sofort aus. Einen unmittelbaren Befehl geben Sie mit einer einzelnen Taste oder Ctrl-Tastenkombination ein. Für eine Ctrl-Tastenkombination halten Sie die Ctrl-Taste gedrückt, während Sie den dazugehörigen Buchstaben eingeben. So bedeutet <Ctrl>-M, daß <Ctrl> gedrückt sein muß, während der Buchstabe »M« eingetippt wird.

Im erweiterten Modus erscheinen alle Eingaben in der Befehlszeile. ED führt keinen Befehl aus, solange Sie die Befehlszeile nicht beenden. Es können eine Reihe von erweiterten Befehlen in einer einzigen Befehlszeile stehen. Jeder Befehl kann ebenso mit anderen zu einer Gruppe zusammengefaßt werden. Die meisten unmittelbaren Befehle haben eine entsprechende erweiterte Version.

ED hält den Bildschirm stets auf dem aktuellen Stand. Geben Sie einen weiteren Befehl ein, während die Anzeige aktualisiert wird, führt ED den Befehl sofort aus, ändert aber die Anzeige bei nächster Gelegenheit. Die aktuelle Zeile wird immer zuerst angezeigt und ist auch immer auf dem neuesten Stand.

3.2 Unmittelbare Befehle

Dieser Abschnitt beschreibt die Befehle, die von ED sofort ausgeführt werden. Unmittelbare Befehle führen folgende Funktionen aus:

- Cursorsteuerung
- Text einfügen
- Text löschen
- Text scrollen
- Befehle wiederholen

3.2.1 Die Benutzung des Cursors

Um den Cursor um ein Zeichen in irgendeine Richtung zu bewegen, betätigen Sie die entsprechende Cursortaste rechts unten auf der Tastatur. Befindet sich der Cursor auf der rechten Seite des Bildschirms, scrollt ED den Text nach links, um den restlichen Text sichtbar zu machen. ED scrollt senkrecht eine Zeile auf einmal und waagerecht zehn Zeichen auf einmal. Der Cursor kann nicht außerhalb des Textes gebracht werden, das heißt, er kann nicht vor den Textanfang, hinter das Textende oder über den linken Textrand hinaus gesetzt werden.

<Ctrl>-], das ist <Ctrl> und die schließende, eckige Klammer »]«, bringen den Cursor auf die rechte Seite der aktuellen Zeile. Nochmalige Eingabe von <Ctrl>-] bringt ihn wieder auf die linke Seite der Zeile zurück. Falls notwendig, wird der Text dazu waagerecht gescrollt. In gleicher Weise bringt <Ctrl>-E den Cursor an den Anfang der ersten Zeile des Bildschirms, falls er sich nicht schon dort befindet. Wenn der Cursor sich bereits dort befindet, bringt ihn <Ctrl>-E an das Ende der letzten Zeile des Bildschirms.

<Ctrl>-T bringt den Cursor an den Anfang des nächsten Wortes. <Ctrl>-R stellt den Cursor auf das Leerzeichen vor dem Wort. In beiden Fällen wird der Text, falls notwendig, senkrecht oder waagerecht gescrollt. Die Tab-Taste bringt den Cursor zur nächsten Tabulator-Position, dem Vielfachen der Tab-Einstellung 3. Es wird kein Tab-Zeichen in die Datei eingefügt.

3.2.2 Text einfügen

Jeder Buchstabe, den Sie im unmittelbaren Modus eintippen, erscheint an der aktuellen Cursorposition, falls die Zeile nicht zu lang ist (eine Zeile darf maximal aus 255 Zeichen bestehen). Versuchen Sie, mehr Zeichen einzugeben, nimmt ED keine weiteren an und gibt die folgende Meldung aus:

```
Line too long
```

In kürzeren Zeilen bewegt ED jedes Zeichen rechts vom Cursor weiter nach rechts, um neuem Text Platz zu machen. Wenn die Zeile den rechten Bildschirmrand erreicht, verschwindet die linke Seite der Zeile. ED zeigt wieder den Zeilenanfang, wenn der Text waagerecht gescrollt wird. Wird der Cursor über das Zeilenende hinaus bewegt, zum Beispiel mit <Tab> oder den Ctrl-Tasten, fügt ED Leerzeichen zwischen dem Zeilenende und jedem neu eingefügten Zeichen ein.

Um die aktuelle Zeile beim Cursor zu trennen und eine neue Zeile zu beginnen, betätigen Sie die Return-Taste. Ist der Cursor am Zeilenende, macht ED nach der aktuellen Zeile eine Leerzeile. Alternativ dazu können Sie mit <Ctrl>-A eine neue Leerzeile nach der gegenwärtigen erzeugen, wobei die aktuelle Zeile nicht aufgeteilt wird. In beiden Fällen erscheint der Cursor am linken Rand der neuen Zeile; voreingestellt ist Spalte 1.

Um sicherzustellen, daß ED automatisch das Zeichen für Wagenrücklauf erzeugt, können Sie einen rechten Rand setzen. Damit beendet ED die Zeile immer vor dem Wort, das über diesen Rand hinausgehen würde, und setzt dieses Wort und den Cursor in die nächste Zeile. Man nennt diesen Vorgang Word-Wrap.

Beachten Sie bitte, daß diese automatische Randeinstellung nicht ordnungsgemäß arbeitet, wenn die Zeile keine Leerzeichen enthält, da ED nicht erkennen kann, wo der Zeilenumbruch stattfinden soll. Wird ein Zeichen eingegeben, und ist der Cursor am rechten Zeilenende, erzeugt ED automatisch eine neue Zeile. War das gerade eingegebene Zeichen kein Leerzeichen, setzt ED das Zeichen am Zeilenende in die neue Zeile. Wird Text innerhalb der Zeile eingefügt, sind also rechts vom Cursor bereits Zeichen vorhanden, arbeitet der automatische Zeilenumbruch nicht! Der rechte Rand ist auf Spalte 79 voreingestellt. Der rechte Rand kann mit dem Befehl EX abgeschaltet werden. Weitere Einzelheiten zum Setzen des Randes finden Sie in Abschnitt 3.3.1 *Programmsteuerung* dieses Kapitels.

Haben Sie Text im falschen Schriftmodus geschrieben, zum Beispiel in Kleinschrift anstatt in Großschrift, kann dies mit <Ctrl>-F korrigiert werden. Bringen Sie den Cursor auf den ersten Buchstaben, der geändert werden soll, und geben Sie dann <Ctrl>-F ein. Wurde der Buchstabe in Kleinschrift geschrieben, ändert ihn <Ctrl>-F in einen Großbuchstaben. Wurde er in Großschrift geschrieben, ändert ihn <Ctrl>-F in einen Kleinbuchstaben. Befindet sich der Cursor auf keinem Buchstaben, zum Beispiel auf einem Leerzeichen oder einer Ziffer, ändert <Ctrl>-F nichts.

<Ctrl>-F setzt den Cursor um ein Zeichen nach rechts, auch wenn er auf ein Leerzeichen trifft. Ist jedoch das nächste Zeichen ein Buchstabe, kann die Groß-/Kleinschreibung mit <Ctrl>-F wiederum verändert werden. Der Befehl kann so lange wiederholt werden, bis alle Zeichen einer Zeile geändert sind. Geben Sie <Ctrl>-F nach dem letzten Buchstaben einer Textzeile wiederholt ein, wandert der Cursor bis zum Ende der Bildschirmzeile, auch wenn nichts verändert werden kann.

Haben Sie diese Zeile:

```
Der Igel und der Hase gehen Hand in Hand
```

und Sie tippen über jedem Zeichen <Ctrl>-F, ändert sich die Zeile folgendermaßen:

```
dER iGEL UND DER hASE GEHEN hAND IN hAND
```

Aus der folgenden Zeile:

```
IF <datei> <= x
```

wird dann:

```
if <DATEI> <= X
```

Alle Buchstaben werden geändert, andere Zeichen und Ziffern aber beibehalten.

3.2.3 Text löschen

Die Backspace-Taste löscht das Zeichen links vom Cursor und bewegt dabei den Cursor um ein Zeichen nach links, wenn er sich nicht am Zeilenanfang befindet. Falls notwendig, scrollt ED dabei den Text. Die Del-Taste löscht das Zeichen unter dem Cursor, ohne daß dieser dabei bewegt wird. Wie bei jedem Löschen, werden Zeichen auf der Zeile nachgezogen und Text, der sich unsichtbar auf der rechten Seite befand, wird sichtbar.

<Ctrl>-O dient zum Löschen von Worten. Die genaue Funktion hängt von dem Zeichen ab, auf dem sich der Cursor befindet. Ist dieses Zeichen ein Leerzeichen, löscht <Ctrl>-O alle Leerzeichen bis zum nächsten Zeichen auf der Zeile, das kein Leerzeichen ist. Anderenfalls löscht es alle Zeichen links der Cursorposition, bis ein Leerzeichen erscheint.

<Ctrl>-Y löscht alle Zeichen von der Cursorposition bis zum Zeilenende.

<Ctrl>-B löscht die ganze aktuelle Zeile. Zum Löschen eines größeren Textblocks können Sie erweiterte Befehle einsetzen.

3.2.4 Scrollen des Textes

Mit den Tasten <Ctrl>-U und <Ctrl>-D wird der Text jeweils 12 Zeilen auf- oder abwärts gescrollt.

<Ctrl>-D läßt den Cursor auf der Zeile, während der Text nach unten gescrollt wird.

<Ctrl>-U scrollt den Text hoch und bewegt den Cursor damit in der Datei 12 Zeilen weiter.

<Ctrl>-V bringt den ursprünglichen Bildschirminhalt zurück. Diese Funktion wird nur gebraucht, wenn ein weiteres Programm neben dem Editor den Bildschirm benutzt und ihn verändert. Im normalen Gebrauch erscheinen Meldungen anderer Programme im hinter dem Editorfenster liegenden Fenster.

3.2.5 Wiederholung von Anweisungen

Der Editor merkt sich jede eingegebene erweiterte Befehlszeile. Um diese Befehle jederzeit wieder auszuführen, brauchen Sie nur <Ctrl>-G einzugeben. Auf diese Weise kann man einen Suchbefehl als einen erweiterten Befehl aufstellen. Entspricht die erste gefundene Zeichenkette nicht der gewünschten, betätigen Sie <Ctrl>-G, um die Suche fortzusetzen. Sie können komplexe Editierbefehle so viele Male ausführen, ohne sie neu eingeben zu müssen.

Geben Sie einen erweiterten Befehl als Befehlsgruppe mit einem Wiederholungszähler an, wiederholt ED die Befehle der Gruppe automatisch mehrmals. Sie brauchen dazu dann nicht <Ctrl>-G zu drücken. Im nächsten Abschnitt finden sich weitere Einzelheiten zu den erweiterten Befehlen.

3.3 Erweiterte Anweisungen

Dieser Abschnitt beschreibt die Befehle, die Ihnen im erweiterten Modus zur Verfügung stehen. Diese Befehle betreffen

- Programmkontrolle
- Blockkontrolle
- Bewegung von Textteilen
- Text suchen
- Text ersetzen
- Text verändern
- Text einfügen

In den erweiterten Modus gelangt man mit der Esc-Taste. Nachfolgende Eingaben erscheinen in der Befehlszeile am unteren Bildschirmrand. Fehler können Sie in der üblichen Weise mit der Backspace-Taste verbessern. Die ganze Zeile beenden Sie mit einem weiteren <Esc> oder mit der Return-Taste. Betätigen Sie <Esc>, verbleibt der Editor im erweiterten Modus, nachdem die Befehlszeile ausgeführt wurde. Drücken Sie dagegen <Return>, kehrt AmigaDOS in den unmittelbaren Modus zurück. Um die Befehlszeile leer zu lassen, geben Sie ein <Return> nach einem <Esc> ein. In diesem Fall kehrt ED in den unmittelbaren Modus zurück.

Erweiterte Befehle bestehen aus einem oder zwei Buchstaben, die in Groß- oder Kleinschrift eingegeben werden können. Sie können auch mehrere Befehle in der gleichen Befehlszeile eingeben, indem Sie sie durch ein Semikolon »;« trennen. Befehle können zusätzliche Argumente wie eine Zahl oder eine Zeichenkette besitzen. Eine Zeichenkette ist eine Folge von Buchstaben, die von einem Sonderzeichen eröffnet und beendet wird. Ein Sonderzeichen

ist jedes Zeichen, das kein Buchstabe ist, zum Beispiel Leerzeichen, Semikolons oder Klammern. Gültige Zeichenketten könnten zum Beispiel so aussehen:

```
/Glück/
!23 Meter!
:Hallo!:
"1/2"
```

Die meisten unmittelbaren Befehle haben eine entsprechende erweiterte Version. Eine vollständige Aufstellung findet sich in der Übersicht der *erweiterten Befehle* am Ende dieses Kapitels.

3.3.1 Programmsteuerung

Dieser Abschnitt beschreibt die Befehle zur Programmkontrolle. Diese Befehle sind: X (eXit), Q (Quit), SA (SAve), U (Undo), SH (SHow), ST (Set Tab), SL und SR (Set Left oder Set Right) und EX (EXtended).

Um die Arbeit mit ED zu beenden, geben Sie den Befehl X ein. Nach der Eingabe dieses Befehls schreibt ED den im Speicher befindlichen Text an ein Ausgabegerät oder in eine Zieldatei. Gibt man die Zieldatei aus, erkennt man, daß alle durchgeführten Änderungen vorhanden sind.

ED legt eine zeitweilige Sicherheitskopie der jeweils letzten Version einer geänderten Datei als :T/ED-BACKUP auf der Diskette ab. Diese Backup-Datei bleibt bestehen, bis man den Bildschirm-Editor verläßt. Dann überschreibt ED die Datei mit einem neuen Backup.

Wollen Sie den Editor verlassen, ohne daß Änderungen gespeichert werden, verwenden Sie den Befehl Q. Nach Q beendet ED die Arbeit unverzüglich, ohne eine Sicherheitskopie anzulegen und ohne die eingegebenen Änderungen abzuspeichern. Aus diesem Grund fragt ED bei geändertem Dateiinhalt nach, ob ein Ausstieg wirklich gewünscht wird.

Ein weiterer Befehl bewirkt, ähnlich einem *Schnappschuß*, eine Sicherung der Datei, wie sie gerade bearbeitet wird. Dies ist der Befehl SA. SA speichert den aktuellen Text in einer benannten Datei oder, falls kein Name angegeben, in der aktuellen Datei ab. Zum Beispiel:

```
SA!:Doc/gespeicherter.Text!
```

oder

```
SA
```

SA sollte immer zur Arbeitssicherung eingesetzt werden, wenn umfangreiche Veränderungen an einem großen Textfile gemacht werden sollen. Nichts ist ärgerlicher als der Verlust vieler Stunden Arbeit, wenn gegen Ende der Strom ausfällt oder sich der Rechner aufhängt und alles löscht. Der Befehl SA gefolgt von Q entspricht übrigens dem Befehl X

Werden irgendwelche Änderungen zwischen den Befehlen SA und Q eingegeben, erscheint folgende Meldung:

```
Edits will be lost - type Y to confirm:
```

»Eingaben würden verlorengehen – geben Sie zur Bestätigung Y ein.« Haben Sie keine Änderungen gemacht, endet ED unmittelbar, der Inhalt der Quelldatei bleibt unverändert. Der Befehl SA ist auch deshalb nützlich, weil er die Eingabe eines anderen Dateinamens als des gegenwärtigen erlaubt. Aus diesem Grund ist es möglich, Kopien von verschiedenen *Entwicklungsstadien* eines Files zu machen und diese in verschiedenen Dateien oder Directories abzulegen.

Um die jeweils letzte Änderung zurückzunehmen, wird der Befehl U verwendet. ED macht intern stets eine Kopie der Zeile, in der sich der Cursor gerade befindet, und ändert diese Kopie, wenn Zeichen hinzugefügt oder gelöscht werden. ED legt diese geänderte Kopie der Zeile erst in der Datei ab, wenn der Cursor aus der aktuellen Zeile bewegt wird. Dies gilt sowohl bei Gebrauch der Cursortasten als auch beim Löschen oder Einfügen einer Zeile. Auch beim waagerechten und senkrechten Scrollen wird die geänderte Zeile übernommen. Der Befehl U löscht die geänderte Kopie und setzt die alte Version der Zeile ein.

Achtung! ED kann keine gelöschte Zeile zurückbringen. Sobald die aktuelle Zeile verlassen wurde, bleibt <Esc>-U wirkungslos.

Der Befehl SH zeigt den aktuellen Status des Editors an. Auf dem Bildschirm werden Informationen ausgegeben, wie die Tabulatorstops, die Randeinstellungen, Blockmarkierungen, der Name der gerade editierten Datei und der belegte Pufferspeicher.

Zu Beginn jeder Sitzung setzt ED in jeder dritten Spalte einen Tabulator. Mit dem Befehl ST gefolgt von einer Zahl n ändern Sie die gegenwärtige Tabulatoreinstellung. ED setzt dann alle n Spalten einen Tabulator.

Den rechten und linken Rand des Textes legen Sie mit den Befehlen SR und SL gefolgt von einer Zahl, welche die gewünschte Spaltenposition bezeichnet, fest. Der linke Rand sollte dabei kleiner als die Bildschirmbreite sein.

Mit dem Befehl EX wird die Randeinstellung außer Kraft gesetzt. Geben Sie EX ein, dann nimmt ED in der aktuellen Zeile auf den rechten Rand keine Rücksicht mehr. Bewegt man den Cursor aus der aktuellen Zeile, aktiviert ED die Randeinstellung wieder.

3.3.2 Blocksteuerung

Soll ein Text bewegt, dupliziert oder gelöscht werden, verwendet man die in diesem Abschnitt erläuterten Blockkontroll-Befehle.

Ein Textblock wird mit den Befehlen BS (Blockanfang) und BE (Blockende) bezeichnet. Dazu bringen Sie den Cursor an eine beliebige Stelle der ersten Zeile des gewünschten Blockes, dort geben Sie den Befehl BS ein. Danach bewegen Sie den Cursor in die Zeile, die die letzte Zeile des Blocks werden soll. Dort gibt man den Befehl BE zum Setzen des Blockendes ein.

Haben Sie mit den Befehlen BS und BE einen Block festgelegt und ändern Sie den Text danach in irgendeiner Weise, sind Blockanfang und -ende nicht mehr definiert. Die einzige Ausnahme von dieser Regel bildet der Befehl IB (Insert Block) zum Einfügen des Textblocks.

Um nur eine Zeile als aktuellen Block zu bezeichnen, bringen Sie den Cursor auf die gewünschte Zeile und geben <Esc> und

```
BS;BE
```

ein. Die aktuelle Zeile wird dann zum aktuellen Block.

Sie können einen Block nicht innerhalb einer Zeile beginnen oder beenden. (Der Block enthält immer ganze Zeilen.) Dazu müssen Sie zunächst die Zeile mit der Return-Taste aufteilen.

Haben Sie einen Block definiert, können Sie eine Kopie davon mit dem Befehl IB (Insert Block) an beliebiger Stelle der Datei in den Text einfügen. Geben Sie diesen Befehl ein, fügt ED eine Kopie des Blocks unmittelbar hinter der aktuellen Zeile ein. Es können mehrere Kopien des Blocks eingefügt werden, solange der Block definiert ist, das heißt bis er verändert oder gelöscht wird.

Zum Löschen eines Blocks wird der Befehl DB (Delete Block) verwendet. DB löscht den mit den Befehlen BS und BE festgelegten Block. Sobald der Block gelöscht ist, gehen auch die Werte für Blockanfang und -ende verloren. Das bedeutet, daß ein Block nicht erst gelöscht und dann eine Kopie davon eingefügt werden kann. Die Befehlsfolge DB;IB bleibt also wirkungslos. Dagegen ist eine Kopie eines Blocks mit folgendem Löschen des Originals durchaus möglich (IB gefolgt von DB).

Mit Blockmarkierungen können Sie Teile der Datei zur späteren Erinnerung markieren. Der Befehl SB (Show Block) zeigt immer die erste Zeile des Blocks in der ersten Bildschirmzeile.

Um einen Block in eine andere Datei zu schreiben, wird der Befehl WB (Write Block) verwendet. Der Befehl benötigt ein Argument, das den Dateinamen bezeichnet. Der folgende Befehl:

```
WB !:Doc/Beispiel!
```

schreibt zum Beispiel den Blockinhalt in die Datei BEISPIEL, die sich im Directory DOC befindet. (Wird der Schrägstrich (/) zum Trennen von Directory und Dateien verwendet, sollte er nicht als Begrenzung für die Zeichenkette verwandt werden!) ED öffnet dazu eine

neue Datei mit dem Namen, den Sie bestimmen. Dabei wird möglicherweise eine bestehende Datei mit gleichem Namen gelöscht und der Pufferinhalt hineingeschrieben!

Mit dem Befehl IF (Insert File) fügen Sie eine Datei in die aktuelle Datei ein. ED liest dazu die Datei in den Speicher, die mit der nach IF eingegebenen Zeichenkette als Argument bezeichnet wurde. Die Datei wird unmittelbar nach der aktuellen Zeile eingefügt. Der folgende Befehl fügt die Datei :DOC/BEISPIEL unmittelbar nach der aktuellen Zeile in die gegenwärtige Datei ein:

```
IF !:Doc/Beispiel!
```

3.3.3 Verändern der Cursorposition

Der Befehl T bringt den Cursor an den Dateianfang, so daß sich die erste Zeile der Datei auf der ersten Bildschirmzeile befindet. Der Befehl B bewegt den Cursor an das Dateiende, so daß sich die letzte Zeile der Datei in der letzten Bildschirmzeile befindet.

Die Befehle N und P bringen den Cursor entsprechend an den Anfang der nächsten Zeile (N) und der vorhergehenden Zeile (P).

Die Befehle CL und CR bewegen den Cursor jeweils ein Zeichen nach links (CL) oder rechts (CR), während CS ihn an den Anfang und CE ans Ende der Zeile bringt.

Der Befehl M bewegt den Cursor zu einer bestimmten Zeile. Nach M wird die Zeilennummer eingegeben, die die aktuelle Zeile werden soll. Der folgende Befehl bringt den Cursor zum Beispiel in Zeile 503 der Datei:

```
M 503
```

Der Befehl M ist eine schnelle Art, um an eine bekannte Position in der Datei zu gelangen. Natürlich kann auch zum Befehl N eine Wiederholungszahl angegeben werden, um in die richtige Zeile der Datei zu gelangen, dies ist aber bedeutend langsamer als die eben beschriebene Methode.

3.3.4 Suchen und Ersetzen von Text

Das Bildschirmfenster kann aber auch mit dem Befehl F (Find) zu einem einzelnen Begriff bewegt werden; dem Befehl F folgt eine Zeichenkette, die den Text bezeichnet, der angesprungen werden soll. Die Suche beginnt ein Zeichen vor der aktuellen Cursorposition und wird bis zum Dateiende fortgesetzt. Wird die Zeichenkette gefunden, so erscheint der Cursor am Anfang der gefundenen Zeichenkette. Die Zeile wird zur ersten Zeile auf dem Bildschirm.

Um den Text rückwärts zu durchsuchen, verwenden Sie den Befehl BF (Backward Find) auf die gleiche Weise wie F. BF findet die letzte vorhandene Zeichenkette vor der aktuellen

Cursorposition. BF sucht links vom Cursor und dann alle Zeilen rückwärts bis zum Dateianfang nach der Zeichenkette.

Um die erste Stelle der Datei zu finden, in der die Zeichenkette auftritt, verwenden Sie T (Top-of-file) gefolgt von F. Die letzte Stelle findet man mit B (Bottom-of-file) gefolgt von BF.

Der Befehl E tauscht zwei durch Abgrenzer getrennte Zeichenketten gegeneinander aus, die erste Zeichenkette, die bereits im Text vorhanden ist, wird durch die zweite ersetzt. So ersetzt zum Beispiel

```
E/Anton/Fritz/
```

das Wort ANTON an der ersten Stelle, an der es auftritt, durch FRITZ. Der Editor beginnt die Suche nach der ersten Zeichenkette an der aktuellen Cursorposition und fährt damit bis zum Dateiende fort. Sobald der Austausch erledigt ist, springt der Cursor zum Ende des ersetzten Textes.

Leere Zeichenketten bezeichnen Sie durch Eingabe von zwei Abgrenzern ohne Zwischenraum. Ist die erste, die *Such*-Zeichenkette im obigen Beispiel leer, fügt der Editor die zweite Zeichenkette an der aktuellen Cursorposition ein. Ist die zweite Zeichenkette leer, bewirkt das ein Löschen der gesuchten Zeichenkette. Der folgende Befehl löscht das nächste Auftreten von ANTON im Text:

```
E/Anton//
```

Während des Austauschens von Textteilen ignoriert ED übrigens alle Randbegrenzungen.

Der Befehl EQ (Exchange and Query) ist eine Variante des E-Befehls. Geben Sie den Befehl EQ ein, fragt ED Sie, ob der Austausch stattfinden soll. Dies ist sinnvoll, wenn nur in einigen Fällen ersetzt werden soll und in anderen nicht. So erscheint nach Eingabe von

```
EQ/Anton/Fritz/
```

folgende Meldung:

```
Exchange?
```

in der Befehlszeile. Wird nun ein »N« getippt, geht der Cursor an der gesuchten Zeichenkette vorüber. Tippen Sie hingegen »Y«, wird wie gewohnt ausgetauscht.

Die Suche-und-ersetze-Befehle machen gewöhnlich während der Suche einen Unterschied zwischen Groß- und Kleinschreibung. Um Groß- und Kleinschreibung zu ignorieren, brauchen Sie nur den Befehl UC zu erteilen. Nach Gebrauch von UC paßt zu der Such-Zeichenkette »Anton« auch »ANTON«, »AntoN« oder »AnTON«. Soll ED wieder zwischen Groß- und Kleinschreibung unterscheiden, können Sie UC durch LC wieder aufheben.

3.3.5 Text ändern

Sie können den Befehl E nicht zum Einfügen einer neuen Zeile in den bestehenden Text verwenden. Statt dessen gibt es dafür die Kommandos I und A. Geben Sie nach dem Befehl I (Insert before) eine Zeichenkette ein, die in den Text eingefügt werden soll. ED fügt diese dann vor der aktuellen Zeile ein. Zum Beispiel:

```
I /Füge dies VOR der aktuellen Zeile ein/
```

setzt »Füge dies VOR der aktuellen Zeile ein« als eine neue, eigenständige Zeile vor die aktuelle Zeile, in der der Cursor steht. Den Befehl A (insert After) verwenden Sie in der gleichen Weise, nur wird die neue Zeile nach der aktuellen eingefügt. So setzt

```
A /Füge dies NACH der aktuellen Zeile ein/
```

die Zeile »Füge dies NACH der aktuellen Zeile ein« als eine neue Zeile nach der aktuellen ein, in der der Cursor steht.

Mit dem Befehl S trennen Sie die aktuelle Zeile an der Cursorposition. Im erweiterten Modus hat S die Funktion der Return-Taste im Direktmodus. In Abschnitt 3.2.2 finden sich weitere Details zum Trennen von Zeilen.

Der Befehl J fügt die nächste Zeile an das Ende der aktuellen an.

Der Befehl D löscht die aktuelle Zeile in der gleicher Weise wie <Ctrl>-B im unmittelbaren Modus. Der Befehl DC löscht das Zeichen auf der Cursorposition ebenso wie die Del-Taste.

3.3.6 Wiederholte Befehle

Um einen Befehl mehrmals zu wiederholen, stellen Sie ihm die Anzahl der Wiederholungen voran. Der Befehl

```
4 E /gehen/wandern
```

ändert die nächsten vier Stellen, an denen »gehen« vorkommt, in »wandern« um. ED bringt den Bildschirm nach jeder Änderung auf den neuesten Stand.

Mit dem Befehl RP (RePeat) wird ein Befehl ausgeführt, bis ein Fehler auftritt – zum Beispiel, weil das Dateiende erreicht ist.

```
T; RP E /gehen/wandern
```

ändert alle Stellen, an denen »gehen« vorkommt, in »wandern« um.

Das Kommando T wird vorangestellt, um sicherzugehen, daß alle vorkommenden Worte »gehen« geändert werden. Wird T weggelassen, werden Änderungen nur ab der aktuellen Cursorposition ausgeführt.

Befehlssequenzen werden wiederholt ausgeführt, wenn sie in runden Klammern zusammengefaßt wurden. Sie können Befehlssequenzen übrigens auch ineinander verschachteln. Der Befehl

```
RP (F /Hilfestellung/; 3 A//)
```

fügt zum Beispiel drei Leerzeilen (Kopien des Leer-Strings) nach jeder Zeile ein, die das Wort »Hilfestellung« enthält.

Beachten Sie bitte, daß diese Befehlszeile nur die Zeilen ab der Cursorposition bis zum Dateiende abarbeitet. Damit die Befehlsfolge die ganze Datei erfaßt, muß vorher an den Dateianfang gesprungen werden.

Einige Befehle sind zwar möglich, aber sinnlos. Zum Beispiel:

```
RP SR 60
```

setzt den Rand ab Spalte 60 und wiederholt diesen Befehl bis in alle Ewigkeit. Jede Folge an erweiterten Befehlen, auch solche, die wiederholt abgearbeitet werden, können Sie jedoch durch Eingabe eines beliebigen Zeichens abbrechen.

3.3.7 Kurzübersicht der ED-Befehle

Sondertasten

Befehl	Funktion
<Backspace>	Löscht das Zeichen links vom Cursor.
<Del>	Löscht das Zeichen an der Cursorposition.
<Esc>	Einschalten des erweiterten Modus.
<Return>	Trennt Zeile an der Cursorposition und erzeugt eine neue Zeile.
<Tab>	Setzt Cursor nach rechts an die nächste Tabulatorposition. (Fügt kein Tab-Zeichen ein!)
<↑>	Bewegt Cursor nach oben.
<↓>	Bewegt Cursor nach unten.
<→>	Bewegt Cursor nach rechts.
<←>	Bewegt Cursor nach links.

Unmittelbare Befehle

Befehl	Funktion
<Ctrl>-A	Fügt eine Zeile ein.
<Ctrl>-B	Löscht eine Zeile.
<Ctrl>-D	Scrollt Text nach unten.
<Ctrl>-E	Setzt Cursor an Bildschirmanfang oder -ende.
<Ctrl>-F	Ändert Groß-/Kleinschreibung.
<Ctrl>-G	Wiederholt die letzte erweiterte Befehlszeile.

Befehl	Funktion
<Ctrl>-H	Löscht das Zeichen links vom Cursor (wie <Backspace>).
<Ctrl>-I	Bewegt Cursor zur nächsten rechten Tabulatorposition.
<Ctrl>-M	Return.
<Ctrl>-O	Löscht Wort oder Leerzeichen.
<Ctrl>-R	Cursor an das Ende des vorhergehenden Wortes.
<Ctrl>-T	Cursor an den Anfang des nächsten Wortes.
<Ctrl>-U	Scrollt Text nach oben.
<Ctrl>-V	Bringt den ursprünglichen Bildschirminhalt zurück.
<Ctrl>-Y	Löscht bis zum Ende der Zeile.
<Ctrl>-[	Einschalten des erweiterten Modus (wie <Esc>).
<Ctrl>-]	Cursor an Zeilenanfang oder -ende setzen.

Erweiterte Befehle

Es folgt eine vollständige Aufstellung der erweiterten Befehle, einschließlich derer, die nur eine erweiterte Version von unmittelbaren sind. In der Aufstellung bedeuten die drei Zeichen /s/, daß dem Befehl eine Zeichenkette folgt. Die Zeichen /s/t/ bedeuten, daß zwei Zeichenketten ausgetauscht werden sollen. »n« steht für eine Zahl.

Befehl	Funktion
A/s/	Einfügen einer Zeile hinter der aktuellen Zeile.
B	Cursor zum Dateiende.
BE	Cursorposition ist Blockende.
BF/s/	Sucht Zeichenkette nach oben.
BS	Cursorposition ist Blockanfang.
CE	Cursor an das Zeilenende setzen.
CL	Cursor um ein Zeichen nach links setzen.
CR	Cursor um ein Zeichen nach rechts setzen.
CS	Cursor an den Zeilenanfang setzen.
D	Löscht die aktuelle Zeile.
DB	Löscht Block.
DC	Löscht Zeichen an der Cursorposition.
E /s/t/	Ersetzt s durch t.
EQ /s/t/	Ersetzt wie E, aber erst nach Rückfrage.
EX	Ignoriert den rechten Rand.
F /s/	Sucht Zeichenkette s.
I /s/	Fügt Zeichenkette vor der aktuellen Zeile ein.
IB	Fügt Kopie eines Blocks vor der aktuellen Zeile ein.
IF /s/	Fügt Datei vor der aktuellen Zeile ein.
J	Verbindet die aktuelle Zeile mit der nächsten.
LC	Beim Suchen zwischen Groß- und Kleinschreibung unterscheiden.

Befehl	**Funktion**
M n	Bewegt den Cursor zur Zeile Nummer n.
N	Bewegt den Cursor zur nächsten Zeile.
P	Bewegt den Cursor zur vorhergehenden Zeile.
Q	ED wird ohne Speicherung des Texts verlassen.
RP	Wiederholt bis zu einem Fehler.
S	Trennt Zeile an Cursorposition.
SA /s/	Speichert Text in eine Datei s.
SB	Zeigt Block auf dem Bildschirm.
SH	Zeigt Informationen über den Status des Editors an.
SL n	Setzt linken Rand an Position n.
SR n	Setzt rechten Rand an Position n.
ST n	Setzt Tabulatorabstand n fest.
T	Cursor zum Dateianfang setzen.
U	Nimmt letzte Eingabe in der aktuellen Zeile zurück.
UC	Beim Suchen Groß- und Kleinschreibung ignorieren.
WB/s/	Schreibt Block in eine Datei s.
X	Verlassen des Programmes ED mit vorherigem Abspeichern des Textes.

Kapitel 4:
EDIT – Der Zeileneditor

Dieses Kapitel beschreibt im einzelnen die Handhabung des Zeileneditors EDIT. Der erste Teil führt den Leser in den Editor ein. Der zweite Teil beschreibt alle Editorfunktionen vollständig. Eine Kurzübersicht über alle EDIT-Befehle findet sich am Ende des Kapitels.

4.1 EDIT – Ein zeilenorientierter Texteditor

EDIT ist ein Texteditor, der sequentielle Dateien Zeile für Zeile abarbeitet. EDIT bewegt sich durch die Eingabe- oder Quelldatei, wobei jede Zeile, nach eventuellen Änderungen, in eine sequentielle Ausgabedatei, die Zieldatei, übernommen wird. Der Ablauf von EDIT erzeugt eine Kopie der Quelldatei, die alle Änderungen enthält, die Sie mit den Editier-Befehlen gefordert haben.

Obwohl EDIT die Ursprungsdatei, auch Quelldatei genannt, üblicherweise Wort für Wort, also sequentiell abarbeitet, läßt sich auch eine begrenzte Zahl von zurückliegenden Zeilen bearbeiten. Dies ist möglich, da EDIT die abgearbeiteten Zeilen nicht sofort in die Zieldatei schreibt, sondern sie statt dessen in einem Ausgabe-Puffer ablegt. Die Größe dieses Puffers hängt vom verfügbaren Speicherplatz ab. Sollen sehr viele Informationen im Speicher abgelegt werden, können Sie mit der EDIT-Option OPT, die im nächsten Abschnitt beschrieben wird, den Speicher vergrößern. Jeder Text kann mehrmals abgearbeitet werden.

Mit den EDIT-Befehlen können Sie:

a) Teile der Quelldatei ändern.
b) Teile der Quelldatei an andere Dateien ausgeben.
c) Material anderer Dateien einfügen.

4.1.1 Aufrufen von EDIT

Dieser Abschnitt beschreibt die Syntax von EDIT und die Parameter, die beim Aufruf an EDIT übergeben werden können oder müssen. EDIT wird mit folgendem Muster aufgerufen:

```
EDIT "FROM/A,TO,WITH/K,VER/K,OPT/K"
```

Das Muster eines Befehls ist in Kapitel 1 dieses Handbuches genauer beschrieben. AmigaDOS erlaubt alle in diesem Muster angegebenen Argumente in der angegebenen Form. Einige der Argumente sind optional. Das heißt, sie werden nur bei Bedarf eingegeben. Einige Argumente ohne »/A« werden von AmigaDOS alleine durch ihre Position richtig zugeordnet. Argumente, denen in der Musterbeschreibung ein »/A« angehängt ist (wie zum Beispiel FROM), müssen beim Aufruf von EDIT angegeben werden. Das zugehörige Schlüsselwort aber kann entfallen. Argumente, denen ein »/K« folgt (zum Beispiel WITH), sind optional, werden sie aber angegeben, muß auch das Schlüsselwort mit erscheinen. Einige müssen mit dem zugehörigen Schlüsselwort erscheinen. Sollten Sie die Syntax von EDIT nicht mehr wissen, genügt es, den folgenden Befehl einzutippen:

```
EDIT ?
```

AmigaDOS zeigt dann das vollständige Muster des Befehls. Das Format von EDIT lautet:

```
EDIT [FROM]<file>[[TO]<file>] [WITH <file>] [VER<file>] [OPT Pn|Wn|PnWn]
```

Nach dem Start von EDIT wird übrigens ein Directory namens »:T« eingerichtet, falls es noch nicht existiert. Dieses Directory dient der Aufnahme temporärer Dateien.

Mehr Informationen über diese Muster und die Syntax von Befehlen finden Sie in den Kapiteln 1 und 2 dieses Handbuchs.

Das Argument FROM deklariert die Quelldatei, die editiert werden soll. Das Argument muß auftreten, das Schlüsselwort selbst kann, muß aber nicht, angegeben werden; das heißt, AmigaDOS erkennt die FROM-Datei an ihrer Position.

Das TO steht für die Zieldatei. Das ist die Datei, in der EDIT die Ausgabe mit allen Änderungen ablegen soll. Wird das TO-Argument weggelassen, benutzt EDIT eine interne Datei, die unter dem Namen der FROM-Datei abgespeichert wird, wenn der Editiervorgang beendet ist. Wird der EDIT-Befehl STOP eingesetzt, findet diese Umbenennung nicht statt, und die FROM-Datei bleibt unverändert erhalten.

Das Schlüsselwort WITH steht für die Datei, welche die Editierbefehle enthält. Lassen Sie das Argument WITH fort, liest EDIT Befehle von der Tastatur.

Das Schlüsselwort VER steht für die Datei, in die EDIT Fehlermeldungen und Zeilenbestätigungen schreibt. Wird VER weggelassen, verwendet EDIT auch hier das Terminal.

Mit dem Schlüsselwort OPT schließlich werden die möglichen Optionen angegeben. Gültige Optionen sind P(n), womit die Zeilenzahl auf die ganzzahlige Variable (n) eingestellt wird, und W(n), sie legt die maximale Zeilenlänge auf (n) Zeichen fest. Falls Sie die Werte nicht ändern, sind sie auf P40, W120 eingestellt. OPT können Sie indirekt zum Vergrößern oder Verkleinern des verfügbaren Speichers verwenden. EDIT verwendet P*W, also die Zahl der maximalen Zeilen multipliziert mit der Zeilenbreite, um den benötigten Speicherplatz festzulegen. Die Speichergröße verändern Sie durch Einstellen der P und W zugeordneten Zahlen. P50 stellt mehr Speicher als üblich zur Verfügung, P30 weniger.

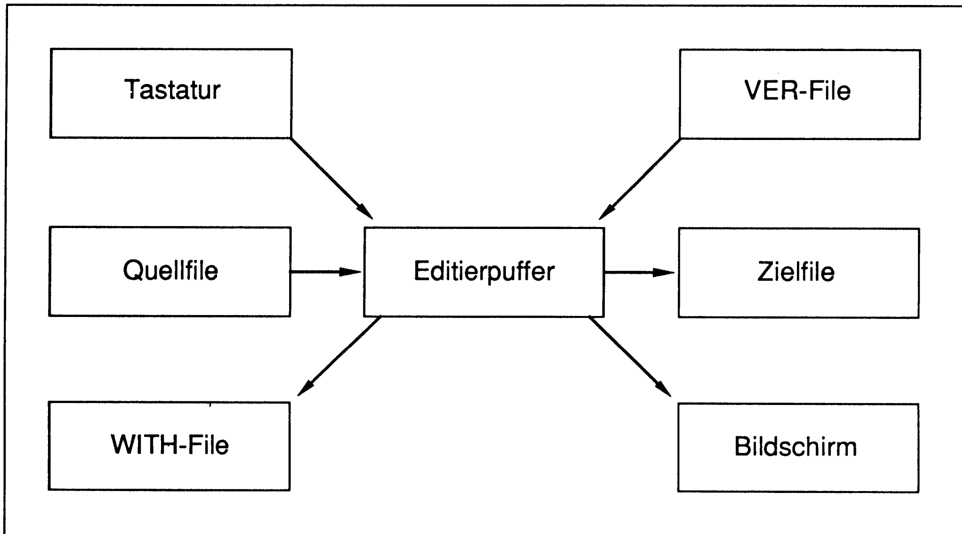


Bild 4.1: *Die Funktionsweise von EDIT*

Hier nun einige Muster-Befehle zum Aufruf von EDIT:

```
EDIT prog1 TO prog1neu WITH editierbfehlen

EDIT prog1 OPT P50W240

EDIT prog1 VER verdatei
```

Achtung! Im Gegensatz zu ED können Sie mit EDIT keine neue Datei anlegen. Versuchen Sie dies, gibt AmigaDOS eine Fehlermeldung aus, weil es die neue Datei im aktuellen Directory nicht finden kann.

4.1.2 EDIT-Kommandos einsetzen

Dieser Abschnitt führt einige der grundlegenden EDIT-Befehle ein, wobei viele der weiterführenden Möglichkeiten weggelassen werden. Eine vollständige Beschreibung der Befehlssyntax erfolgt in Abschnitt 4.2 dieses Kapitels.

4.1.2.1 Die aktuelle Zeile

Wenn EDIT von der Quelldatei Zeilen einliest und sie dann in die Zieldatei schreibt, wird die Zeile, die gerade bearbeitet wird, als die aktuelle Zeile bezeichnet. EDIT führt alle Textänderungen in der aktuellen Zeile aus. EDIT fügt neue Zeilen stets vor die aktuelle Zeile ein. Wird EDIT gestartet, wird die erste Zeile der Quelldatei die aktuelle Zeile.

4.1.2.2 Zeilennummern

EDIT verbindet jede Zeile der Quelldatei mit einer Zeilennummer. Diese Zeilennummer ist kein Teil der in der Datei gespeicherten Information, sondern EDIT weist sie durch Zählen der eingelesenen Zeilen zu. Bei der Arbeit mit EDIT können Sie auf die Zeilennummer verweisen. Eine eingelesene Zeile behält, solange sie sich im Hauptspeicher befindet, ihre Zeilennummer bei, auch wenn Zeilen vor oder nach ihr gelöscht oder eingefügt werden. Die Zeilennummern bleiben unverändert, bis Sie die Arbeit beenden oder die Zeilen mit dem Befehl »=« umnumerieren. Immer wenn Sie eine Datei aufrufen, weist EDIT Zeilennummern zu. Beim wiederholten Aufruf der Datei unterscheiden sich die Zeilennummern aber unter Umständen.

4.1.2.3 Auswahl der aktuellen Zeile

Eine aktuelle Zeile können Sie auf drei Arten ansprechen:

a) durch Abzählen der Zeilen

b) durch Angeben des Inhaltes

c) durch Angeben einer Zeilennummer

Diese drei Methoden werden nun beschrieben:

Abzählen der Zeilen:

Die Befehle N und P erlauben den Sprung zur nächsten oder vorhergehenden Zeile. Geben Sie vor diesen Befehlen eine Nummer an, können Sie diese Anzahl an Zeilen vorwärts oder rückwärts springen. Um eine Zeile vorwärts zu springen, müssen Sie zum Beispiel eingeben:

```
N
```

Jeden EDIT-Befehl können Sie sowohl in Großschrift als auch in Kleinschrift eingeben.

Um vier Zeilen vorwärts zu springen, geben Sie ein:

```
4 N
```

Die vierte Zeile nach der aktuellen wird zur neuen aktuellen Zeile.

Um eine Zeile zurückzuspringen, tippen Sie:

```
P
```

Der Befehl P erlaubt ebenfalls die Angabe von Zahlen. Geben Sie zum Beispiel den folgenden Befehl ein, wird die vierte Zeile vor der aktuellen Zeile zur neuen aktuellen Zeile:

```
4 P
```

Sie können nur zu Zeilen zurückspringen, die EDIT noch nicht in die Ausgabedatei geschrieben hat. EDIT erlaubt normalerweise nur ein Zurückspringen um 40 Zeilen. Um weiter zurückspringen zu können, müssen Sie beim Aufruf von EDIT mit der Option P mehr Zeilen einrichten. Dazu finden sich weitere Einzelheiten im Abschnitt 4.1.1 dieses Kapitels.

Eine spezielle Zeilennummer anspringen:

Der Befehl M erlaubt, durch Angabe der Zeilennummer eine neue aktuelle Zeile anzuwählen. Dazu werden der Befehl M und die gewünschte Zeilennummer eingegeben. So weist zum Beispiel der Befehl M 45 (Move 45) EDIT an, zu Zeile 45 zu springen. Befindet man sich hinter der Zeile 45, so wird auch zu ihr zurückgesprungen, vorausgesetzt, daß sie sich noch im Speicher befindet.

Die Angabe einer Zeilennummer und das Abzählen der Zeilen können miteinander kombiniert werden. Zum Beispiel:

```
M12; 3N
```

springt zu Zeile15.

Aufeinanderfolgende Befehle in der gleichen Zeile trennen Sie durch ein Semikolon (;).

Auswahl einer Zeile durch ihren Inhalt:

Mit dem Befehl F (Find) können Sie eine aktuelle Zeile durch ihren Inhalt aufsuchen lassen. Zum Beispiel:

```
F/Inhalt/
```

findet die nächste Zeile, in der sich der Begriff »Inhalt« befindet. Die Suche beginnt bei der aktuellen Zeile und geht vorwärts durch die gesamte Quelldatei, bis die benötigte Zeile gefunden ist. Falls EDIT das Dateiende erreicht hat, ohne eine passende Zeile zu finden, wird folgende Meldung ausgegeben:

```
INPUT EXHAUSTED
```

Ebenso ist es möglich, mit dem Befehl BF (Backwards Find) rückwärts zu suchen:

```
BF/Zucker und Zimt/
```

Die Suche beginnt in der aktuellen Zeile, EDIT bewegt sich nun aber rückwärts in der Datei, bis die gewünschte Zeile gefunden ist. Erreicht EDIT den Anfang der Datei, ohne daß eine passende Zeile gefunden wird, erscheint folgende Meldung:

```
NO MORE PREVIOUS LINES
```

Beachten Sie bitte, daß in den obigen Beispielen der gesuchte Text zwischen einzelnen Schrägstrichen (/) steht. Dieser angegebene Text wird als Zeichenkette bezeichnet. Die Zeichen, die Sie zum Anzeigen des Anfangs und des Endes einer Zeichenkette verwenden, werden Abgrenzungszeichen genannt. Einige spezielle Zeichen, wie : . , und * sind als Abgrenzer verwendbar; natürlich darf die Zeichenkette selbst keine Abgrenzer enthalten. EDIT ignoriert die Leerzeichen zwischen Befehl und dem ersten Abgrenzer, beachtet aber Leerzeichen innerhalb der Zeichenkette, der Inhalt wird genauestens verglichen. Zum Beispiel:

```
F /Da Da Baum/
```

findet »Da-Da Baum« oder »Da Da-Baum« nicht.

F ohne Argument wiederholt die vorhergehende Suchoperation. Der folgende Befehl zum Beispiel findet die zweite Zeile im File, in der »grauer Vogel« steht.

```
F/grauer Vogel/; N; F
```

Der Befehl N zwischen den beiden F-Befehlen ist notwendig, da F immer in der aktuellen Zeile mit der Suche beginnt. Lassen Sie N fort, findet das zweite F die gleiche Stelle in der aktuellen Zeile wie das erste.

4.1.2.4 Qualifikatoren

Die oben beschriebene, grundlegende Form des Befehls F findet eine Zeile, die die angegebene Zeichenkette an irgendeiner Stelle enthält. Um die Suche auf den Anfang oder das Ende einer Zeile zu beschränken, stellen Sie dem Suchbegriff den Buchstaben B (für Beginn) oder E (für Ende) voran. In diesem Fall müssen Sie nach dem F ein oder mehrere Leerzeichen eingeben.

Der folgende Befehl :

```
F B /glückliche Zeiten/
```

findet eine Zeile, die mit »glückliche Zeiten« beginnt, während

```
F E /Hosenträger/
```

eine Zeile findet, die mit »Hosenträger« endet. Werden dem gesuchten Inhalt genauere Bezeichnungen angehängt, beschleunigt sich die Suche, da EDIT nur einen Teil jeder Zeile durchsuchen muß.

In der oben angegebenen Form sind B und E Beispiele für Qualifikatoren, und das ganze Argument wird als *qualifizierte Zeichenkette* bezeichnet. Weitere Qualifikatoren stehen zur Verfügung. Zum Beispiel der Qualifikator P:

```
F P /Unter den Linden/
```

bedeutet, daß eine Zeile gesucht wird, die ausschließlich den Text »Unter den Linden« enthält. Die gewünschte Zeile darf keine weiteren Zeichen vor oder nach der angegebenen Zeichenkette enthalten. Das heißt, nach Eingabe dieses Befehls findet EDIT die Zeile:

```
Unter den Linden
```

Die folgende Zeile jedoch wird nicht gefunden:

```
Unter den Linden.
```

Um eine leere Zeile zu finden, können Sie eine leere Zeichenkette mit dem Qualifikator P verwenden. Zum Beispiel:

```
F P//
```

Sie können mehrere Qualifikatoren in beliebiger Reihenfolge gebrauchen.

4.1.2.5 Ändern der aktuellen Zeile

Dieser Abschnitt beschreibt den Einsatz der Befehle E, A und B zum Ändern des Texts in der aktuellen Zeile. Der folgende Befehl zum Beispiel:

```
E/Wunderland/Fernglas/
```

ersetzt die Zeichenkette »Wunderland« durch den Begriff »Fernglas«. Beachten Sie bitte, daß Sie in der Mitte nur einen einzelnen Abgrenzer zum Trennen der beiden Zeichenketten benutzen. Um Teile des Textes zu löschen, das heißt, Text durch nichts zu ersetzen, geben Sie eine leere zweite Zeichenkette wie folgt an:

```
E/Gitarre//
```

Um einer Zeile neuen Text anzufügen, verwenden Sie die Befehle A oder B. Das Kommando A fügt die zweite Zeichenkette nach der ersten ein. In gleicher Weise fügt der Befehl B die zweite Zeichenkette vor dem ersten Erscheinen der ersten ein. Wenn zum Beispiel die aktuelle Zeile den Text

```
acht ist gleich acht
```

enthält und die folgende Befehlsfolge eingegeben wird:

```
A/acht/zig/; B L /acht/zweiundsiebzig und /
```

ändert sich der Text in den folgenden:

```
achtzig ist gleich zweiundsiebzig und acht
```

Würden Sie den Qualifikator L bei dem Befehl B wegelassen, sähe das Ergebnis so aus:

```
zweiundsiebzig und achtzig ist gleich acht
```

da von links nach rechts gesucht wird und EDIT bei der ersten gefundenen Stelle beginnt. Der Qualifikator L bedeutet, daß der Befehl am Zeilenende mit der Suche beginnen soll.

Ist das erste Argument in einem A-, B- oder E-Befehl leer, fügt EDIT die zweite Zeichenkette an den Anfang oder das Ende einer Zeile ein. Um die Position der zweiten Zeichenkette näher zu bezeichnen, verwenden Sie die Qualifikatoren L oder E oder lassen sie weg.

Geben Sie einen A-, B- oder E-Befehl ein, der den Suchbegriff nicht findet, erscheint folgende Meldung entweder auf dem Bildschirm oder in der VER-Datei, sofern Sie eine beim Aufrufen von EDIT angegeben haben:

```
NO MATCH
```

Weitere Einzelheiten zur VER-Datei finden Sie im Abschnitt 4.1.1 *Aufrufen von EDIT.*

4.1.2.6 Löschen ganzer Zeilen

Dieser Abschnitt beschreibt, wie Zeilen aus der Datei entfernt werden können. Um eine Reihe von Zeilen zu löschen, können Sie ihre Zeilennummern in einem D-Befehl verwenden. Sie wenden den Befehl D an, indem Sie D und die Zeilennummer eingeben. Geben Sie nach der Zeilennummer ein Leerzeichen und eine zweite Zahl ein, löscht EDIT alle Zeilen von der ersten Zeilennummer bis zur letzten. Zum Beispiel:

```
D97 104
```

löscht die Zeilen 97 bis 104 einschließlich, Zeile 105 ist dann die neue aktuelle Zeile. Die aktuelle Zeile löschen Sie durch Eingabe von D ohne weitere Angaben.

Der folgende Befehl:

```
F /Erdbeereis/; D
```

löscht zum Beispiel die Zeile, die »Erdbeereis« enthält, und die darauf folgende Zeile wird zur aktuellen. Sie können eine qualifizierte Suche mit dem Befehl »D« wie folgt kombinieren:

```
F B/Die/; 4D
```

Diese Befehlsfolge löscht vier nacheinander folgende Zeilen, deren erste Zeile mit »Die« beginnt.

Statt einer Zeilennummer können Sie auch einen Punkt (.) oder einen Stern (*) eingeben. Mit dem Punkt verweisen Sie auf die aktuelle Zeile. Auf das Dateiende verweisen Sie mit dem Stern. Der folgende Befehl löscht zum Beispiel den restlichen Text der Quelldatei einschließlich der aktuellen Zeile:

```
D . *
```

4.1.2.7 Einfügen neuer Zeilen

Dieser Abschnitt beschreibt, wie mit Hilfe von EDIT neue Zeilen in die Datei eingefügt werden. Um eine oder mehrere Zeilen **vor** einer angegebenen Zeile einzufügen, verwenden Sie den Befehl I (Insert). Sie können den Befehl I einzeln oder in Verbindung mit einer Zeilennummer, einem Punkt (.) oder einem Stern (*) eingeben. EDIT fügt den Text vor der aktuellen Zeile ein, wenn Sie nur I oder I zusammen mit einem Punkt (.) angeben. Geben Sie nach I einen Stern (*) ein, wird der Text am Dateiende angefügt; das heißt nach der letzten Zeile der Datei. Jeder neu eingegebene Text wird vor der angegebenen Zeile eingefügt.

Das Ende des eingefügten Textes zeigen Sie an, indem Sie <Return> betätigen, Z eingeben und nochmals <Return> drücken. Zum Beispiel:

```
I 468
Die kleinen Fische im Meer,
antworten Dir nimmer mehr.
Z
```

fügt die beiden Textzeilen vor Zeile 468 ein.

Lassen Sie die Zeilennummer nach dem Befehl fort, fügt EDIT den neuen Text vor der aktuellen Zeile ein. Zum Beispiel:

```
F /Wecker/; I
Er sagte: "Ich werde ihn aufwecken gehen, wenn..."
Z
```

Dieser Mehrfach-Befehl sucht die Zeile, die den Text »Wecker« enthält und die damit auch zur aktuellen Zeile wird, und fügt die dahinter angegebene neue Zeile ein.

Nach dem I-Befehl mit einer Zeilennummer ist die aktuelle Zeile stets die Zeile mit der angegebenen Nummer; anderenfalls bleibt die aktuelle Zeile unverändert.

Um langes Eintippen zu ersparen, bietet EDIT den Befehl R (Replace), der genau der Eingabe von DI, also D für Löschen und I für Einfügen, entspricht. Zum Beispiel:

```
R 19 26
Wenn im Winter die Felder weiß sind
Z
```

löscht die Zeilen 19 bis einschließlich 26 und fügt dann den neuen Text vor Zeile 27, die nun die aktuelle Zeile geworden ist, ein.

4.1.2.8 Wiederholen von Befehlen

Viele EDIT-Befehle können automatisch wiederholt werden, wie oben in den Beispielen zu N und D gezeigt wurde. Zudem kann eine Gruppe von Befehlen wiederholt werden, indem

sie in Klammern gesetzt zu einer Befehlsgruppe formiert wird. Der folgende Befehl löscht die nächsten sechs leeren Zeilen in der Quelldatei:

```
6 (F P//; D)
```

Befehlsgruppen dürfen nur eine Zeile lang sein.

4.1.3 Verlassen von EDIT

Um die Arbeit mit EDIT zu beenden, setzen Sie den Befehl W (Wind up) ein. EDIT spult sich bis zum Dateiende der Quelldatei durch, kopiert diese in die Zieldatei und steigt aus. Geben Sie keine mit TO angegebene Datei an, benennt EDIT die interne Ausgabedatei mit dem Namen der FROM-Datei.

EDIT kann Befehle aus verschiedenen Quellen erhalten. Im einfachsten Fall nimmt EDIT die Befehle direkt von der Tastatur; dies wird als primäre Befehlsstufe bezeichnet. EDIT kann darüber hinaus Befehle von anderen Quellen, wie zum Beispiel Befehlsdateien oder auch WITH-Dateien, annehmen.

Mit dem Befehl C können Sie Befehlsdateien während der Arbeit von EDIT aufrufen, innerhalb der Befehlsdateien können Sie dann weitere aufrufen, so daß jede verschachtelte Befehlsdatei zu einer eigenständigen Befehlsstufe wird. EDIT stoppt die Ausführung der Befehle in der Befehlsdatei, wenn das Dateiende erreicht wird oder der Befehl Q gefunden wurde. Findet EDIT ein Q in einer Befehlsdatei oder wird das Dateiende erreicht, bricht EDIT die Ausführung der Befehle der Datei ab und kehrt zur vorhergehenden Befehlsstufe zurück. Findet EDIT einen Q-Befehl in einer verschachtelten Befehlsdatei, kehrt EDIT zur Ausführung der Befehle in der Befehlsdatei, die jeweils eine Stufe höher liegt, zurück. Beenden Sie das Editieren auf der primären Befehlsstufe durch Eingabe eines Q-Befehls oder findet EDIT in einer WITH-Datei ein Q, dann beendet EDIT die Arbeit in gleicher Weise, als ob Sie den Befehl W verwendet hätten.

Der Befehl STOP beendet EDIT sofort, also ohne weiteres Abarbeiten der Datei. Dabei ist besonders zu beachten, daß EDIT keine der noch im Speicher stehenden Zeilen in die Zieldatei schreibt. Deklarieren Sie nur das FROM-Argument, überschreibt EDIT die Quelldatei nicht mit der unvollständig editierten Datei. Der Befehl STOP sollte nur dann verwendet werden, wenn das veränderte File nicht gebraucht wird.

EDIT legt eine interne Sicherheitskopie der bearbeiteten Datei in der Datei :T/ED-BACKUP ab, wenn Sie die Arbeit mit den Befehlen W oder Q beenden. Diese Backup-Datei verbleibt im Speicher, bis EDIT ein zweites Mal aufgerufen und wieder beendet wird. Dann überschreibt EDIT die Datei mit dem neuen Backup. Verwenden Sie den Befehl STOP, wird diese Datei nicht erstellt.

4.1.4 Ein kombiniertes Beispiel: alle Befehle auf einmal

Mit den bisher beschriebenen Befehlen können alle wichtigen Arbeiten innerhalb eines Files durchgeführt werden. Dieser Abschnitt nun stellt ein Beispiel vor, das einige dieser Befehle verwendet. Der Text, der im Beispiel den Editier-Befehlen in Kursivschrift folgt, ist Kommentar. Diese Kommentare dürfen nicht mit eingegeben werden; EDIT läßt in Befehlszeilen keine Kommentare zu.

Tippen Sie bitte folgenden Quelltext ab. Verwenden Sie den ED, die Zeilennummern bitte nicht mit eingeben:

```
1 Früher, wo ich sehr unerfahren
2 und bescheid´ner als heute
3 Hatten meine höchste achtung
4
5 Später tref ich auf den Weide
6 außer mir noch and´re Kälber
7 und nun schätz´ ich, sozusagen
8 erst dich selber.
```

Führen Sie nun folgende Änderungen durch (die Reihenfolge der Befehle ist wichtig!):

```
M E/wo/da/ ;E /sehr //            <1.Zeile, »wo« durch »da« ersetzen; »sehr« < löschen>
N; A /bescheid´ner /war /;A L//,/ <nächste Zeile, »war« einsetzen; nach dem
                                  <ersten Leerzeichen von rechts ein Komma
                                  <anfügen>
N; E /H/h/; E L /a/A/             <das erste »H« durch »h« ersetzen, das letzte
                                  <»a« durch »A« ersetzen.
F P//; I                          <vor der Leerzeile einsetzen>
and´re Leute.
Z
N; E /tref/traf/; E /den/der/     <Nächste Zeile; Teile austauschen>
M6; 2(A L//,/; N)                 <Zu Zeile 6; das letzte Leerzeichen durch ein
                                  <»,« ersetzen; eine Zeile weiter; dort ebenfalls
                                  <ersetzen>
F B/erst/; E/d/m/                 <Zeile suchen, die mit »erst« beginnt; »d«
                                  <durch »m« ersetzen>
W                                 <geändertes File speichern, EDIT beenden>
```

Man erhält den folgenden Text (mit neuen Zeilennummern):

```
1 Früher, da ich unerfahren
2 und bescheid'ner war als heute,
3 hatten meine höchste Achtung and're Leute.
4
5 Später traf ich auf der Weide
6 außer mir noch and're Kälber,
7 und nun schätz' ich, sozusagen,
8 erst mich selber.
```

Experimentieren Sie ruhig mit diesem Quelltext. Dabei werden Sie feststellen, daß es verschiedene Wege zu einer Lösung gibt. (Die Verse stammen übrigens von Wilhelm Busch.) So kann zum Beispiel die letzte Zeile auch so eingegeben werden:

```
E/d/m/
```

Das Resultat bleibt gleich, da diese Zeile durch das »N« in der vorhergehenden schon zur aktuellen wurde.

4.2 Vollständige Beschreibung des Zeileneditors

Im ersten Abschnitt dieses Kapitels haben Sie sich mit den grundlegenden Funktionen von EDIT vertraut gemacht. Damit können Sie schon alle normalerweise vorkommenden Arbeiten mit EDIT durchführen. Dieses Kapitel enthält eine vollständige Beschreibung aller Möglichkeiten von EDIT. Diesen Abschnitt werden Sie nur brauchen, wenn Probleme mit EDIT auftauchen oder Sie sehr komplizierte Änderungen an einer Datei vornehmen wollen.

Im einzelnen wird beschrieben:

- Befehlssyntax
- Befehlsfiles, Quellfile und Ausgabefile
- Arbeiten mit EDIT
- Funktionelle Gruppen von EDIT-Befehlen
- Zeilenfenster
- String-Bearbeitung in der aktuellen Zeile
- Andere Befehle für die aktuelle Zeile
- Teile des Quellcodes lesen
- Schleifen
- Umfassende Eingriffe
- Anzeigen des Programmstatus
- Beenden eines EDIT-Ablaufes
- Verändern der aktuellen Zeile
- Abbrechen der interaktiven EDITierung

4.2.1 Befehlssyntax

EDIT-Befehle bestehen aus einem Befehlsnamen, ohne oder mit einem oder mehreren Argumenten. Ein oder mehrere Leerzeichen zwischen Befehlsnamen und dem ersten Argument, zwischen Argumenten ohne Zeichenketten und zwischen Befehlen sind erlaubt. Ein Leerzeichen zwischen diesen Argumenten ist nur notwendig, um aufeinanderfolgende zu trennen, die sonst als eine Einheit behandelt werden; zum Beispiel zwei Zahlen.

EDIT erkennt, daß ein Befehl beendet ist:

- wenn RETURN gedrückt wird
- wenn EDIT das Ende der Argumente eines Befehls erreicht
- wenn EDIT auf ein Semikolon (;) stößt
- wenn EDIT auf das Zeichen »)« stößt

Klammern setzen Sie, um Befehlsgruppen zu bilden.

Befehle, die in einer Eingabezeile auftreten, trennen Sie durch ein Semikolon (;). Dies ist nur notwendig, wenn ein Befehl eine variable Anzahl an Argumenten besitzt. EDIT nimmt immer die ausführlichste Form eines Befehls an. Sie können Befehle in Groß- und Kleinschrift eingeben.

4.2.1.1 Befehlsnamen

Ein Befehlsname ist entweder eine Reihe von Buchstaben oder ein einzelnes, spezielles Zeichen, wie zum Beispiel #. Ein alphabetischer Befehlsname endet mit einem Zeichen, das kein Buchstabe ist. Nur die ersten vier Buchstaben des Namens werden berücksichtigt. Ein oder mehrere Leerzeichen können zwischen dem Befehlsnamen und seinen Argumenten auftreten. EDIT verlangt mindestens ein Leerzeichen, wenn ein Argument, das mit einem Buchstaben beginnt, einem alphanumerischen Namen folgt.

4.2.1.2 Argumente

Die folgenden Abschnitte beschreiben die sechs verschiedenen Arten von Argumenten, die Sie zusammen mit EDIT-Befehlen verwenden können:

- Zeichenketten
- qualifizierte Zeichenketten
- Suchausdrücke
- Zahlen
- Einstellwerte
- Befehlsgruppen

4.2.1.3 Zeichenketten

Eine Zeichenkette ist eine Folge von bis zu 80 Zeichen, die von Abgrenzern eingeschlossen sind. Leere (Null-)Zeichenketten sind erlaubt (eine Null-Zeichenkette ist eine Zeichenkette ohne Zeichen, das heißt, die Abgrenzer schließen nichts ein, wie zum Beispiel //). Zeichen,

die Sie zum Begrenzen einer Zeichenkette verwenden, dürfen innerhalb dieser Zeichenkette selbst nicht auftreten. Der beendende Abgrenzer kann weggelassen werden, wenn die Befehlszeile unmittelbar darauf endet.

Folgende Zeichen können als Abgrenzer verwendet werden:

/ • . + - , ? : *

Dies sind also die üblichen Interpunktionszeichen, außer dem Semikolon (;), und die vier arithmetischen Operatoren.

Hier einige Beispiele für Zeichenketten:

```
/A/
*Isartal-Sperre*
??
+Zeichenkette ohne abschließenden Abgrenzer
```

4.2.1.4 Mehrfach-Zeichenketten

Befehle mit zwei Zeichenketten als Argumente müssen den gleichen Abgrenzer für beide Zeichenketten verwenden. Der Abgrenzer zwischen den beiden Argumenten wird nur einmal eingegeben. Hierzu ein Beispiel mit dem Befehl A:

```
A /König/Der Rote/
```

In diesem Befehl deklariert die zweite Zeichenkette den einzufügenden Text. Lassen Sie diese Zeichenkette wegfallen, setzt EDIT eine Null-Zeichenkette ein. Tun Sie dies bei den Befehlen A und B, geschieht nichts, da EDIT angewiesen wird, nichts nach oder vor der ersten Zeichenkette einzufügen. Lassen Sie dagegen die zweite Zeichenkette nach einem E-Befehl fortfallen, löscht EDIT die erste Zeichenkette.

4.2.1.5 Qualifizierte Zeichenketten

Befehle, die nach Bezügen suchen, entweder in der aktuellen Zeile oder in der gesamten Quelldatei, bezeichnen den Bezug durch qualifizierte Zeichenketten. Eine qualifizierte Zeichenkette ist eine Zeichenkette, der kein, ein oder mehrere Qualifikatoren vorausgehen. Die Qualifikatoren sind einzelne Zeichen, die in jeder Reihenfolge verwendet werden können. Zum Beispiel:

```
BU /Abc/
```

Leerzeichen brauchen Sie zwischen den Qualifikatoren nicht einzusetzen. Eine Liste von Qualifikatoren beenden Sie durch ein beliebiges Abgrenzer-Zeichen. Die verfügbaren Qualifikatoren sind:

```
B(Anfang), E(Ende), L(Links oder Letztes), P(Vorher) und U(Großschrift)
```

4.2.1.6 Suchausdrücke

Befehle, die nach einer einzelnen Zeile in der Quelldatei suchen, erwarten einen Suchbegriff als Argument. Ein Suchbegriff ist eine qualifizierte Zeichenkette. Der folgende Befehl sucht zum Beispiel eine Zeile, die mit der Zeichenkette »Weltall« beginnt:

```
F B/Weltall/
```

4.2.1.7 Zahlen

Eine Zahl ist eine Folge dezimaler Ziffern. Zeilennummern sind eine spezielle Zahlenform und müssen stets größer als null sein. Tritt eine Zeilennummer auf, können statt dessen auch die Zeichen ».« und »*« erscheinen. Ein Punkt steht für die aktuelle Zeile, ein Stern für die letzte Zeile der Quelldatei. Der folgende Befehl weist EDIT zum Beispiel an, zum Ende der Quelldatei zu springen:

```
M*
```

4.2.1.8 Einstellwerte

Befehle, die EDIT-Einstellungen verändern, erwarten ein einzelnes Zeichen als Argument. Dieses Zeichen muß ein Plus (+) oder ein Minus (–) sein.

```
V-
```

Das Minus (–) in diesem Befehl weist EDIT an, die Verifizierung einzustellen. Wird dann »V+« eingegeben, schaltet EDIT die Verifizierung wieder an. In diesem Fall können »+« und »–« als »an« und »aus« verstanden werden.

4.2.1.9 Befehlsgruppen

Um eine Anzahl eigenständiger EDIT-Befehle zu einer Befehlsgruppe zusammenzufassen, können Sie die Befehle in Klammern setzen. Die folgende Zeile sucht zum Beispiel die Zeile, in der der Begriff »Adler« erscheint, und ersetzt ihn durch »Großer Vogel«.

```
(F/Adler/; E/Adler/Großer Vogel/)
```

Befehlsgruppen können nicht länger als eine Eingabezeile sein. Geben Sie mehr Befehle ein, nimmt EDIT nur die bis zum Zeilenende an. EDIT findet dann die geschlossene Klammer am Ende der Zeile nicht und gibt die Fehlermeldung aus:

```
Unmatched parenthesis
```

Beachten Sie bitte, daß Sie Klammern nur verwenden müssen, wenn eine Befehlsgruppe mehr als einmal ausgeführt werden soll.

4.2.1.10 Wiederholung der Befehle

Vielen Befehlen kann eine Zahl vorangestellt werden, die EDIT anweist, diesen Befehl zu wiederholen. Die Zahl darf kein Vorzeichen haben und nur aus dezimalen Ziffern bestehen.

Zum Beispiel:

```
24N
```

Geben Sie die Ziffer Null ein, führt EDIT den Befehl unendlich oft aus, das heißt, natürlich nur, bis das Dateiende erreicht ist. Der folgende Befehl ersetzt zum Beispiel bis zum Dateiende jedes »Dum« durch »Dee«:

```
0(E /Dum/Dee/;N)
```

Auch Befehlsgruppen können in gleicher Weise wie einzelne Befehle wiederholt werden:

```
0(F/hadlich/; E/hadlich/handlich/)
```

4.2.2 Programmablauf

Dieser Abschnitt beschreibt die Abläufe bei der Arbeit mit EDIT. Er gibt Auskunft darüber, woher Eingaben kommen und wohin Ausgaben gehen, was auf dem Bildschirm erscheint und was nach dem Ablauf von EDIT in der bearbeiteten Datei steht.

4.2.2.1 Das Eingabe-Zeichen (Prompt)

Arbeiten Sie interaktiv mit EDIT, erscheint ein Prompt, sobald EDIT bereit ist, neue Befehle auszuführen. Interaktiv wird gearbeitet, wenn Befehle von der Tastatur kommen und Ausgaben auf den Bildschirm geschrieben werden. Hat der letzte Befehl eine Ausgabe auf den Bildschirm verursacht, gibt EDIT keinen Prompt aus.

Ist die Verifizierung aktiviert, aktualisiert EDIT die aktuelle Zeile,
- wenn die aktuelle Zeile nicht bereits verifiziert wurde
- wenn seit der letzten Verifizierung in der Zeile irgendwelche Änderungen gemacht wurden
- wenn die Lage des aktiven Fensters verändert wurde.

Wird die aktuelle Zeile von EDIT nicht verifiziert, zeigt der Zeileneditor einen Doppelpunkt (:), um darauf hinzuweisen, daß er zur Ausführung neuer Befehle bereit ist. Dieser Doppelpunkt ist das Eingabe-Zeichen von EDIT. EDIT gibt jedoch keine Prompts aus, wenn Sie Zeilen einfügen.

4.2.2.2 Die aktuelle Zeile

Während EDIT Zeilen aus einer Quelldatei liest oder sie in eine Zieldatei schreibt, wird die Zeile, die gerade bearbeitet wird, zur aktuellen Zeile. Jeder eingegebene Befehl arbeitet mit der aktuellen Zeile. EDIT fügt neue Zeilen vor der aktuellen Zeile ein. Beginnt man mit dem Editieren mit EDIT, ist zunächst die erste Zeile der Quelldatei die aktuelle Zeile.

4.2.2.3 Zeilennummern

EDIT erkennt jede Zeile in der Quelldatei anhand einer nur einmal vergebenen Zeilennummer. Diese ist kein Teil der in der Datei gespeicherten Informationen. EDIT vergibt

diese Nummern durch Abzählen der Zeilen während des Einlesens. Neu eingefügte Zeilen erhalten keine Zeilennummern.

EDIT unterscheidet zwischen Originalzeilen aus dem Quelltext und Nicht-Originalzeilen. Originalzeilen sind Zeilen aus dem Quelltext, die nicht verändert oder getrennt wurden. Nicht-Originalzeilen wurden neu eingefügt oder durch Trennen erzeugt. Befehle, die Zeilennummern als Argument haben, können nur auf die Quellzeilen verweisen. EDIT springt vorwärts und bis zu einer bestimmten Grenze rückwärts. Die Richtung hängt davon ab, ob die eingegebene Zeilennummer größer oder kleiner als die aktuelle ist. EDIT überspringt oder löscht (auf Wunsch) neu eingegebene Zeilen auf der Suche nach angegebenen Quellzeilen.

Gegen Sie an Stelle einer Zeilennummer einen Punkt ein, beziehen sich alle Befehle auf die aktuelle Zeile, ob Quellzeile oder nicht. Ein Beispiel dazu finden Sie in Abschnitt 4.1.2.6 *Löschen ganzer Zeilen.*

Mit dem Befehl »=« werden die Zeilen umnumeriert. Dadurch erhalten neu eingegebene Zeilen Zeilennummern, werden also Quellzeilen gleichgestellt. Der folgende Befehl ändert zum Beispiel die aktuelle Zeilennummer auf 15, alle folgenden bis zum Dateiende werden angepaßt.

```
=15
```

Wird nach dem Befehl »=« keine Nummer angegeben, so zeigt EDIT folgende Meldung an:

```
Number expected after =
```

4.2.2.4 Qualifizierte Zeichenketten

Um den Bereich anzugeben, in dem EDIT suchen soll, verwenden Sie qualifizierte Zeichenketten. EDIT akzeptiert die Null-Zeichenkette, es vergleicht nur mit der angegebenen Suchposition. Die Suchposition ist, falls nicht anders angegeben, der Anfang einer Zeile. Geben Sie keine zusätzlichen Qualifikatoren an, findet EDIT die Zeichenkette an einer beliebigen Stelle in einer Zeile. Qualifikatoren deklarieren zusätzliche Bedingungen zur Eingrenzung der Suche. EDIT erlaubt die fünf Qualifikatoren B, E, L, P und U:

B sucht nur am Anfang der Zeile. Dieser Qualifikator darf nicht zusammen mit E, L oder P auftreten.

E sucht am Ende der Zeile. Dieser Qualifikator darf nicht mit B, L oder P auftreten. Erscheint E mit einer Null-Zeichenkette, entspricht das dem Zeilenende. Das heißt, das Leerzeichen am Zeilenende wird gesucht.

L durchsucht eine Zeile entgegen der normalen Richtung von rechts nach links. Tritt eine Zeichenkette in einer Zeile mehrmals auf, stellt dieser Qualifikator sicher, daß die letzte Zeichenkette statt der ersten gefunden wird. L darf nicht zusammen mit B, E oder P verwendet werden. Geben Sie L mit einer Null-Zeichenkette an, wird das Leerzeichen am Ende der Zeile gefunden.

P sucht nach der Zeile, die nur den als Argument angegebenen String enthält. P darf nicht mit B, E oder L auftreten. P mit einer Null-Zeichenkette als Argument sucht eine Leerzeile.

U setzt Groß- und Kleinschrift gleich; das heißt, die Zeichenkette kann aus jeder Kombination von Groß- und Kleinbuchstaben bestehen. Wird U zum Beispiel mit folgender Zeichenkette verwendet:

```
/HUMPAtäterä/
```

werden auch die beiden folgenden Zeile gefunden:

```
humpaTÄTERÄ
HUMpatätERÄ
```

4.2.2.5 Abspeichern der editierten Zeilen

EDIT schreibt bearbeitete Zeilen nicht unmittelbar in die Zieldatei, sondern speichert sie in einem als Puffer definierten Bereich im Hauptspeicher ab. Hat EDIT den verfügbaren Speicher verbraucht, schreibt es die ersten Zeilen aus diesem Puffer der Reihe nach in die Zieldatei. Bis zu dem Zeitpunkt, an dem EDIT eine bestimmte Zeile in die Zieldatei schreibt, kann man auf sie zurückgreifen und sie wieder zur aktuellen Zeile machen.

Teile der Ausgabe können Sie übrigens auch in andere Zieldateien als in die hinter TO angegebene schreiben.

4.2.2.6 Handhabung des Dateiendes

Stößt EDIT auf das Ende der Quelldatei, wird eine fiktive Datei-Endzeile erzeugt. Diese Datei-Endzeile trägt die nächsthöhere Zeilennummer. Diese Zeile erscheint mit der Zeilennummer »*« auf dem Bildschirm.

Befehle zum Ändern dieser aktuellen Zeile oder Sprünge zu einer weiteren Zeile erzeugen eine Fehlermeldung. Im Gegensatz dazu gibt EDIT keine Fehlermeldung aus, wenn das Dateiende während der Ausführung einer unendlich wiederholten Gruppe erreicht wird.

4.2.3 Funktionelle Einteilung der EDIT-Befehle

Dieser Abschnitt enthält, nach ihrer Funktion geordnet, alle EDIT-Befehle. Eine Zusammenfassung und eine alphabetische Liste folgen am Ende des Kapitels.

Die folgenden Beschreibungen verwenden Schrägstriche (/) als Abgrenzer; Abgrenzer sind Zeichen zum Begrenzen von Zeichenketten. Befehlsnamen werden in Großschrift dargestellt. Die Argumente werden in Kleinschrift angegeben, wie in der folgenden Tabelle beschrieben:

Bezeichnung	Beschreibung
a,b	Zeilennummer (oder ».« oder »*«)
cg	Befehlsgruppen
m,n	Nummern
q	Qualifikatoren-Liste (evtl. leer)
se	Suchausdruck
s,t	Ketten einzelner Zeichen (Zeichenketten)
sw	Einstellwert (+ oder –)
/	Zeichenketten-Abgrenzer

Tabelle 4.1: *Variable zur Beschreibung von Befehlen*

Befehlsbeschreibungen, die die hier aufgelisteten Bezeichnungen verwenden, zeigen die **Syntax** eines Befehls. Sie sind keine Beispiele für das, was Sie eingeben müssen.

4.2.3.1 Auswahl einer aktuellen Zeile

Diese Befehle haben keine weitere Funktion als die Bestimmung einer neuen aktuellen Zeile. EDIT schreibt diese Zeile in den Pufferspeicher, sobald sie bearbeitet wurde. Weitere Informationen zu diesem Thema finden Sie in Abschnitt 4.1 *EDIT - Ein zeilenorientierter Texteditor.*

M nimmt eine Zeilennummer, einen Punkt oder einen Stern als Argument an. Wird in diesem Fall die oben beschriebene Notation verwendet, lautet die korrekte Syntax also:

```
Ma
```

Ma springt in der Quelldatei zur Zeile »a« vor oder zurück. Es können nur Quellzeilen angesprochen werden.

```
M+
```

läßt die letzte von der Datei gelesenen Zeile zur aktuellen werden. M+ liest alle Zeilen, die momentan im Speicher gehalten werden, und macht die letzte zur aktuellen Zeile.

```
M-
```

macht die letzte Zeile im Pufferspeicher zur aktuellen Zeile. EDIT geht also soweit wie möglich zurück.

```
N
```

springt zur nächsten Zeile der Quelldatei vor. Ist die aktuelle Zeile die letzte Zeile der Quelldatei, erscheint keine Fehlermeldung. Die Datei-Endzeile wird angezeigt. Geben Sie dann wiederum den Befehl N ein, gibt EDIT eine Fehlermeldung aus.

```
P
```

springt in die vorhergehende Zeile. Sie können mehrere Zeilen zurückspringen, indem Sie P wiederholt oder vor einer Zahl angeben. Die Zahl muß der Anzahl an Zeilennummern entsprechen, die zurückgesprungen werden soll.

F

findet die gesuchte Zeile durch den Suchausdruck »se«. Die Suche beginnt in der aktuellen Zeile und geht vorwärts durch die gesamte Quelldatei. Ein F-Befehl ohne Argument verwendet den bei der letzten Suchoperation angegebenen Suchbegriff. Die Syntax für den Befehl F lautet:

```
F se
```

Die Syntax für BF (finde rückwärts) lautet:

```
BF se
```

BF verhält sich wie F, sucht jedoch, ausgehend von der aktuellen Zeile, zum Anfang der Quelldatei, bis er eine Zeile findet, die den gesuchten Ausdruck enthält.

4.2.3.2 Einfügen und Löschen einer Zeile

Befehle können neben ihrer Hauptfunktion eine neue aktuelle Zeile bestimmen. Solche Befehle, gefolgt von einzufügendem Text, müssen die letzten Befehle einer Befehlszeile sein. Der einzufügende Text muß sich in den Zeilen unter dem Befehl befinden und wird mit »Z« auf einer eigenen Zeile beendet. Der Befehl Z wird auch zum Ändern des Zeichens für Dateiende verwendet. EDIT nimmt das Zeichen in Großschrift und Kleinschrift an. Der folgende Befehl fügt zum Beispiel den neuen Text vor »a« ein.

```
Ia
(einzufügender Text, soviele Zeilen wie nötig)
Z
```

Zur Erinnerung: »a« kann eine Zeilennummer, ein Punkt (für die aktuelle Zeile) oder ein Stern (für die letzte Zeile der Quelldatei) sein. Lassen Sie »a« fort, fügt EDIT den neuen Text vor der aktuellen Zeile ein; a wird dann die neue aktuelle Zeile.

```
I/s/
```

fügt den Inhalt der Datei »s« (s steht für Zeichenkette) vor der aktuellen Zeile ein.

```
Ra b
(zu ersetzender Text)
Z
```

Der Befehl R entspricht D;I. Die zweite Zeilennummer muß größer oder gleich der ersten sein. Lassen Sie die zweite Zeilennummer weg, ersetzt EDIT nur eine Zeile. Das heißt, b=a.

Werden beide Nummern weggelassen, wird die aktuelle Zeile ersetzt. Die Zeile nach b wird zur neuen aktuellen Zeile.

Der Befehl D (Delete) löscht alle Zeilen von »a« bis einschließlich »b«. Wird die zweite Zeilennummer weggelassen, wird nur die angegebene Zeile gelöscht; das heißt, b=a. Werden beide Nummern weggelassen, wird die aktuelle Zeile gelöscht. Die Zeile, die auf »b« folgt, wird zur neuen aktuellen Zeile. Die Syntax für D lautet:

```
Da b
```

Der Befehl DF weist EDIT an, alle Zeilen ab der aktuellen Zeile zu löschen, bis die Zeile mit dem gesuchten Ausdruck gefunden ist. Diese Zeile wird dann zur neuen aktuellen Zeile. Ein DF-Befehl ohne Argument sucht den zuletzt eingegebenen Suchbegriff.

Die Syntax des Befehls DF (Delete and Find) lautet:

```
DF se
```

4.2.4 Zeilenfenster

EDIT bearbeitet normalerweise eine komplette aktuelle Zeile. Darüber hinaus können aber auch Zeilenteile definiert werden, die dann als Zeilenfenster bezeichnet werden. Dieser Abschnitt beschreibt, wie die Befehle zum Definieren eines solchen Fensters eingesetzt werden.

4.2.4.1 Das operationale Fenster

EDIT untersucht gewöhnlich alle Zeichen einer Zeile, wenn es nach einem String sucht. Es kann jedoch ein Zeilenfenster definiert werden, so daß die Suche nach einem Zeichen am Anfang des *Fensters* und nicht am Anfang einer Zeile beginnt. In allen Beschreibungen von EDIT-Befehlen bedeutet *der Anfang der Zeile* stets auch *der Anfang des operationalen Fensters*, sofern ein solches Fenster definiert ist.

Wenn EDIT eine aktuelle Zeile neu aufbaut, weist es mit einem Zeiger, dem Größer-Zeichen »>«, auf den Beginn des operationalen Fensters der aktuellen Zeile hin. Zum Beispiel:

```
26.
Dies ist Zeile 26.
    >
```

Das operationale Fenster enthält die Zeichen rechts vom Zeiger: »ist Zeile 26.«. EDIT läßt den Indikator weg, falls es sich am Anfang der Zeile befindet.

Die folgenden Befehle zum Bewegen des Fensters stehen zur Verfügung:

```
>
```

bewegt den Zeiger um ein Zeichen nach rechts.

`<`

bewegt den Zeiger um ein Zeichen nach links.

`PR`

Der POINTER RESET setzt den Zeiger an den Anfang der Zeile.

`PA q/s/`

Der Befehl POINT AFTER setzt den Zeiger so, daß das erste Zeichen im Fenster das erste Zeichen nach der Zeichenkette s ist.

`PA L//`

bringt beispielsweise den Zeiger zum Zeilenende.

`PB`

Der Befehl PB (POINT BEFORE) ähnelt dem Befehl PA, nur daß die Zeichenkette selbst im Fenster steht. Die Syntax für PB lautet:

`PB q/s/`

4.2.4.2 Verändern einzelner Zeichen in der aktuellen Zeile

Die folgenden Befehle bewegen den Zeiger ein Zeichen nach rechts, nachdem der erste Buchstabe in einen Groß- oder Kleinbuchstaben verwandelt wurde. Ist das erste Zeichen kein Buchstabe oder steht er bereits in der gewünschten Schriftart, entsprechen diese Befehle »>«.

$ ändert zu Kleinschrift.

% ändert zu Großschrift.

`_`

Der Befehl _ (Unterstreichungsstrich) macht das erste Zeichen im Fenster zu einem Leerzeichen und bewegt den Zeiger eine Stelle nach rechts.

`#`

löscht das erste Zeichen im Fenster. Der im Fenster verbleibende Rest rutscht ein Zeichen nach links, während der Zeiger stehenbleibt und auf das nächste Zeichen der Zeile verweist. Dieser Befehl entspricht genau dem folgenden (wobei »s« das erste Zeichen des Fensters ist):

`E/s//`

Um diesen Befehl zu wiederholen, kann auch eine Zahl vorangestellt werden:

```
5#
```

löscht die nächsten fünf Zeichen im Fenster, entspricht also der Eingabe von »#####«. Geben Sie eine Zahl ein, die größer oder gleich der im Fenster vorhandenen Zeichen ist, löscht EDIT den Inhalt des gesamten Fensters.

Kombinationen aus >, %, $, _ und #-Befehlen können Sie zum zeichenweisen Editieren verwenden. Die Befehle erscheinen unter den Zeichen, die sie betreffen. Dazu folgendes Beispiel:

```
ein männLEIN stehttim walde,, ganz    stilllund STUMM
```

Dies Zeile wird nach Eingabe der folgende Befehle

```
%»%»$$$$»»_»%»>>#»>>#»»_»>$$$$$
```

zu

```
Ein Männlein steht im Walde, ganz still und stumm
```

Der Zeiger bleibt danach unter dem letzten »m« stehen.

4.2.5 Verändern von Zeichenketten in der aktuellen Zeile

Änderungen in der aktuellen Zeile werden mit den grundlegenden Befehlen zur String-Bearbeitung oder mit einer Variante, dem *Zeiger zu*-Befehl, durchgeführt. Beide werden in den nächsten zwei Abschnitten beschrieben.

4.2.5.1 Grundlegende Operationen mit Zeichenketten

Zum Verändern von Teilen der aktuellen Zeile stehen drei sehr ähnliche Befehle zur Verfügung. Die Befehle A, B und E setzen ihr zweites Argument nach (A=After), vor (B=Before) oder anstelle (E=Exchange) des ersten Arguments. Das erste Argument kann zusätzlich durch einen Qualifikator bestimmt sein. Lautet die aktuelle Zeile:

```
Der Bäcker backt Kuchen
```

und Sie geben dann ein:

```
E U/Bäcker/Vater/    Austausch
B/Kuchen/gute /      Einfügen der Zeichenkette vor
A L//;/              Einfügen der Zeichenkette nach
```

ändert sich die Zeile zu

```
Der Vater backt gute Kuchen;
```

4.2.5.2 Der Null-String

Die leere oder Null-Zeichenkette (//) kann nach jedem Zeichenketten-Befehl stehen. Geben Sie die Null-Zeichenkette als zweite Zeichenkette eines E-Befehls ein, löscht EDIT die erste Zeichenkette aus der Zeile. Findet EDIT die erste Zeichenkette, bewirkt die Null-Zeichenkette als zweites Argument bei den Befehlen A oder B nichts; anderenfalls wird eine Fehlermeldung ausgegeben. Bei allen Befehlen entspricht die Eingabe der Null-Zeichenkette als erstes Argument der ersten Suchposition. Die ursprüngliche Suchposition ist die aktuelle Zeigerposition (zu Beginn der Zeilenanfang), wenn nicht die Qualifikatoren E oder L eingegeben wurden. Zum Beispiel:

```
A//Bäcker/
```

setzt den Text »Bäcker« nach »Nichts,« das heißt, an den Anfang der Zeile. Der folgende Befehl

```
A L//Bäcker
```

setzt zum Beispiel den String »Bäcker« an das Ende der Zeile, sozusagen nach dem letzten »Nichts«.

4.2.5.3 Die Zeiger-Variante

Die Befehle AP (insert After and Point), BP (insert Before and Point) und EP (Exchange and Point) haben zwei Zeichenketten als Argumente und arbeiten wie A, B und E. Dabei bleibt der Zeiger jedoch an der letzten Stelle der beiden Argumente stehen. So entspricht:

```
AP/s/t/
```

der Folge

```
A/s/t/; PA/st/
```

während

```
BP/s/t/
```

der Folge

```
B/s/t/; PA/ts/
```

entspricht.

Das folgende Beispiel

```
2EP U/diedel/Diddel/
```

ändert den Text

```
diedeldie und DIEDELdum
```

in

`Diddeldie und Diddeldum`

wobei der Zeiger vor »dum« stehen bleibt.

4.2.5.4 Löschen von Teilen der aktuellen Zeile

Die beiden Befehle DTA (Delete Till After) und DTB (Delete Till Before) löschen vom Zeilenanfang (oder vom Zeiger) bis zu einer angegebenen Zeichenkette. Um von einem angegebenen Punkt zum Zeilenende zu löschen, verwenden Sie die Befehle DFA (Delete From After) und DFB (Delete From Before). Lautet die aktuelle Zeile zum Beispiel:

```
All des Königs Pferde und all des Königs Mannen
```

und der Befehl

```
DTB L/Königs/
```

wird getippt, ändert sich der Text in

```
Königs Mannen
```

Der folgende Befehl hingegen

```
DTA/Pferde /
```

ändert den Text in

```
und all des Königs Mannen
```

4.2.6 Weitere Befehle für die aktuelle Zeile

In diesem Abschnitt wird erklärt, wie Befehle wiederholt werden, wie die aktuelle Zeile aufgetrennt und Zeilen miteinander verbunden werden.

Immer wenn EDIT einen Befehl zum Ändern einer Zeichenkette ausführt (zum Beispiel A, B oder E), wird diese Zeile zur aktuellen. Um den Befehl ein weiteres Mal auszuführen, geben Sie ein Apostroph (´) ein. Dieser Befehl hat kein Argument. Er verwendet die Argumente des letzten A-, B- oder E-Befehls.

> **Achtung!** Verwenden Sie Befehle wie diese, können unerwartete Dinge geschehen:
> `E/Schloß/Ritter/; 4(´; E/Bauer/Königin/)`

Die zweite und weitere Ausführungen des Befehls » ´« verweisen auf eine andere als die erste Ausführung. Das obige Beispiel tauscht zweimal »Schloß« und »Ritter« und siebenmal »Bauer« und »Königin«, anstatt »Schloß« und »Ritter« einmal auszutauschen und danach viermal »Schloß« durch »Ritter« und »Bauer« durch »Königin« zu ersetzen.

4.2.6.1 Trennen und Verbinden von Zeilen

EDIT ist in erster Linie ein Zeileneditor. Die meisten EDIT-Befehle wirken nicht über die Zeilengrenzen hinaus. Dieser Abschnitt beschreibt nun Befehle, die eine Zeile aufteilen oder zwei oder mehr aufeinanderfolgende Zeilen miteinander verbinden.

Um eine Zeile vor einem angegebenen Bezugspunkt aufzuteilen, wird der Befehl SB verwendet. Die Syntax für SB lautet:

```
SB q/s/
```

SB erkennt Qualifikatoren, wie hier durch q dargestellt, und eine Zeichenkette /s/. SB teilt die aktuelle Zeile vor dem Bezugspunkt, den Sie durch Qualifikatoren und die Zeichenkette angeben. EDIT schreibt den ersten Teil der Zeile in den Puffer und macht den verbliebenen Rest zur neuen aktuellen Zeile, die jedoch keine Quellzeile ist.

Um eine Zeile nach dem Bezugspunkt aufzuteilen, wird der Befehl SA verwendet. Die Syntax von SA ist:

```
SA q/s/
```

SA akzeptiert bei Bedarf Qualifikatoren q und eine Zeichenkette /s/. SA teilt die aktuelle Zeile nach dem Bezugspunkt, den Sie durch Qualifikatoren und Zeichenkette angeben.

Um zwei Zeilen zu verketten, verwenden Sie den Befehl CL. Dessen Syntax lautet:

```
CL/s/
```

CL bezieht bei Bedarf eine Zeichenkette, die oben als /s/ dargestellt wird, ein. CL (Concatenate Line) bildet eine neue aktuelle Zeile durch Verbinden der aktuellen Zeile mit der angegebenen Zeichenkette und der nächsten Zeile aus der Quelldatei, und zwar in dieser Reihenfolge. Ist die gesuchte Zeichenkette ein Null-String, braucht sie nicht angegeben werden. Das Aufteilen und Verbinden von Zeilen soll anhand eines Beispiels aufgezeigt werden:

```
Fischers Fritz aß frische Fische; Frische
Fische aß Fischers
Fritz.
```

Dieser Vers ist etwas verstümmelt. Die Zeilen müssen noch in Form gebracht werden. Ist die erste Zeile die aktuelle, ändern die folgenden Befehle

```
SA /;  /; 2CL/ /
```

die erste Zeile zu

```
Fischers Fritz aß frische Fische;
```

und machen die zweite (die nun `Frische Fische aß Fischers Fritz` enthält) zur aktuellen Zeile.

4.2.7 Teile der Quelldatei lesen

Alle folgenden Befehle lesen Zeilen von der Quelldatei und schreiben die dabei abgearbeiteten Zeilen in den Pufferspeicher oder die normale Ausgabedatei. Da diese Befehle in der Regel interaktiv angewandt werden, das heißt, der Bildschirm wird aktualisiert, werden sie *Type-Befehle* genannt. Sie können jedoch auch zur Ausgabe in eine Datei verwandt werden. Nachdem EDIT einen dieser Befehle ausgeführt hat, wird die zuletzt ge-*tippte* Zeile, das heißt die zuletzt auf dem Bildschirm angezeigte, zur neuen aktuellen Zeile.

Die Syntax des Befehls T (Type) lautet:

```
Tn
```

Tn tippt n Zeilen. Lassen Sie n fort, wird bis zum Ende der Quelldatei ausgegeben. Die Ausgabe können Sie jederzeit mit CTRL-C unterbrechen.

Hinweis: Steht irgendwo im Handbuch ein Bindestrich zwischen zwei Tasten, bedeutet das, daß sie zur gleichen Zeit gedrückt werden müssen. CTRL-C heißt deshalb, daß C eingegeben wird, während CTRL gedrückt ist.

Die erste Zeile, die EDIT nach »T« ausgibt, ist die aktuelle. Der folgende Befehl gibt, zum Beispiel beginnend mit der nächsten Zeile, die »Sein oder nicht sein« enthält, sechs Zeilen auf den Bildschirm aus:

```
F /Sein oder nicht sein/; T6
```

Beachten Sie bitte, daß Sie das »S« vom ersten »Sein« in Großschrift eingeben müssen, wenn die richtige Zeile gefunden werden soll.

Der Befehl TP gibt die Zeilen im Pufferspeicher aus. TP (Type Previous) entspricht dem in EDIT ausgeführten Befehl M, gefolgt von der Ausgabe der Zeilen bis zur aktuellen. Die Syntax lautet:

```
TP
```

Der Befehl TN gibt alle Zeilen aus, die sich noch im Hauptspeicher des EDIT befinden. TN (Type Next) gibt alle Zeilen auf dem Bildschirm aus, die auch mit dem Befehl P erreicht werden können. Mehr über den Befehl P finden Sie in Abschnitt 4.1.1 dieses Handbuchs. Der Vorteil des TN-Befehls liegt darin, daß alle ausgegebenen Zeilen mit den Befehlen P und BF angesprungen werden können.

Die Syntax des Befehls TL (Type with Line numbers) lautet:

```
TL n
```

TLn tippt n Zeilen wie T, im Unterschied dazu werden Zeilennummern hinzugefügt. Da eingefügte und aufgeteilte Zeilen keine Zeilennummern haben, gibt EDIT bei ihnen »++++« aus.

Zum Beispiel:

```
20  O Bruder Montague, komm heraus
++++ wir leben hier in Saus und Braus
```

Die Quellzeile beginnt mit »O Bruder« und hat eine Zeilennummer. Die nicht zur Quelldatei gehörende eingefügte Zeile beginnt mit »++++«. Zur Erinnerung: Mit dem Befehl »=« können Sie Zeilen numerieren, auch wenn sie nicht zur Quelldatei gehören.

4.2.8 Verwaltung von Befehls-, Eingabe- und Ausgabedateien

EDIT verwendet vier Dateiarten:

- Befehlsdateien
- Eingabedateien
- Ausgabedateien
- Verifizierungsdateien

Ist EDIT gestartet, kann die Verifizierungsdatei mit keinem Befehl verändert werden. Mehr über die Verifizierungsdatei finden Sie in Abschnitt 4.1.1 dieses Handbuchs. Der folgende Abschnitt beschreibt Befehle, die Befehls-, Eingabe- und Ausgabedateien verändern, die Sie beim Start von EDIT erstellt oder geöffnet haben.

4.2.8.1 Befehlsdateien

EDIT liest Befehle vom Terminal oder aus einer Datei, die Sie mit der Option WITH angegeben haben. Um Befehle aus einer anderen Datei zu lesen, verwenden Sie den Befehl C. Laut Muster lautet die Syntax für den Befehl C:

```
C /s/
```

wobei die Zeichenkette »s« für einen Dateinamen steht. Da AmigaDOS den Schrägstrich (/) zum Trennen von Dateinamen benutzt, sollten Sie Punkte oder andere Symbole als Begrenzer verwenden. Ein Symbol, das sich in der Zeichenkette befindet, sollte nicht als Abgrenzer benutzt werden. Hat EDIT alle Befehle einer Datei ausgeführt (oder ist Q gefunden worden), schließt EDIT die Datei und greift wieder auf den Befehl zurück, der dem Befehl C folgt. Die folgende Befehlszeile zum Beispiel liest und führt die Befehle in der Datei :T/XYZ aus:

```
C .:t/xyz.
```

4.2.8.2 Eingabedateien

Um den gesamten Inhalt einer Datei an einer bestimmten Stelle in die Quelldatei einzufügen, verwenden Sie die Befehle I und R. Diese Befehle wurden bereits in Abschnitt 4.1.2.7 dieses Kapitels besprochen.

In Abschnitt 4.1.1 wird auch beschrieben, wie EDIT aufgerufen wird. Dort wird die Quelldatei als FROM-Datei angegeben. Mit FROM als Schlüsselwort können Sie jedoch auch eine Datei angeben, aus der weitere Zeilen wie Quelltext eingelesen werden sollen. FROM hat folgende Syntax:

```
FROM /s/
```

wobei die Zeichenkette »s« ein Dateiname ist. FROM ohne Argument liest die ursprüngliche Quelldatei noch einmal ein. Wird FROM ausführt, bleibt die aktuelle Zeile gleich, die nächste Zeile kommt aus der neuen Quelldatei.

EDIT schließt eine Datei nicht, wenn die Datei nicht mehr aktuell ist, es können weitere Zeilen aus dieser Quelldatei gelesen werden, indem sie wieder angewählt wird.

Um eine Ausgabedatei, die in EDIT geöffnet wurde, zu schließen, um sie daraufhin zur Eingabe zu öffnen, müssen Sie CF (Close File) verwenden. CF hat folgende Syntax:

```
CF /s/
```

Dabei stellt »s« wieder einen Dateinamen dar. Wird die Arbeit mit EDIT beendet, werden alle Dateien geschlossen.

Öffnen Sie eine Datei, so beginnt EDIT am Anfang dieser Datei zu lesen oder zu schreiben. Wird eine Datei mit CF geschlossen und wieder geöffnet, so beginnt EDIT wieder mit der ersten Zeile der Datei und nicht mit der Zeile, an der EDIT sich befand, als die Datei geschlossen wurde.

Ein Beispiel zum Binden von Zeilen aus zwei Dateien mit dem Befehl FROM:

Befehl	**Wirkung**
M10	arbeitet die Zeilen 1 bis 9 aus der Quelldatei ab.
FROM .XYZ.	wählt neue Eingabe an, Zeile 10 bleibt aktuell.
M6	arbeitet Zeile 10 aus FROM ab und holt die Zeilen 1 bis 5 aus XYZ.
FROM	Datei FROM wird wieder angewählt.
M14	arbeitet Zeile 6 aus XYZ und Zeile 11 bis 13 aus FROM ab.
FROM .XYZ.	Datei FROM wird wieder angewählt.
M*	arbeitet Zeile 14 aus FROM und den Rest aus XYZ ab.
FROM	Datei FROM wird wieder angewählt.
CF .XYZ.	XYZ wird geschlossen.
M*	der Rest von FROM (Zeile 15 bis Dateiende) wird abgearbeitet.

4.2.8.3 Ausgabedateien

EDIT schickt seine Ausgabe in eine TO-Datei. Die Ausgabe erfolgt jedoch nicht unmittelbar. EDIT hält eine bestimmte Anzahl von Zeilen in einem Pufferbereich im Hauptspeicher. Diese Zeilen sind vorhergehende aktuelle Zeilen oder Zeilen, die EDIT bereits abgearbeitet hat, bevor es die gegenwärtige aktuelle Zeile erreichte. Die Anzahl der Zeilen, die EDIT im Speicher behalten kann, hängt von der Puffergröße ab, die Sie beim

Aufruf von EDIT zugewiesen haben. Weil EDIT diese Zeilen zurückhält, hat es die Fähigkeit, in der Quelldatei rückwärts zu gehen.

Um die Ausgabe in eine andere als die beim Start angegebene Datei zu schreiben, wird ebenfalls der Befehl TO verwendet. TO hat die Syntax:

```
TO .s.
```

Dabei ist »s« der Dateiname.

Führt EDIT einen TO-Befehl aus, schreibt es den Pufferspeicher in die jeweilige Datei, wenn die Ausgabedatei geöffnet ist.

EDIT schließt eine Ausgabedatei nicht, wenn sie nicht weiter aktuell ist. Wird die Datei wieder angewählt, können weitere Zeilen hineingeschrieben werden. Das folgende Beispiel zeigt, wie Sie die Ausgabe zwischen der Hauptrichtung TO und einer anderen Richtung XYZ aufteilen können.

Befehl	**Wirkung**
M11	schreibt die Zeilen 1 bis 10 in Richtung TO.
TO.XYZ.	aktiviert die Ausgabedatei XYZ.
M21	schreibt die Zeilen 11 bis 21 in Richtung XYZ.
TO	
M31	schreibt die Zeilen 21 bis 30 in Richtung TO.
TO.XYZ.	
M41	schreibt die Zeilen 31 bis 40 in Richtung XYZ.
TO	

Wollen Sie eine Datei erneut benutzen, so müssen Sie sie vorher schließen. Der folgende Befehl schließt die Datei mit dem als Argument angegebenen Dateinamen:

```
CF .Dateiname.
```

Diese Ein-/Ausgabe-Befehle werden angewandt, wenn ein Teil der Quelldatei von *oben* nach *unten* bewegt werden soll. Die folgende Befehlsliste zeigt dies am Beispiel:

Befehl	**Wirkung**
TO .:T/1.	Ausgabe an eine Hilfs-Datei :T/1.
1000N	schreibt 1000 Zeilen in die Hilfs-Datei.
TO	kehrt zur eigentlichen Ausgabedatei zurück.
CF .:T/1.	schließt Hilfs-Datei :T/1.
I2000.:T/1.	öffnet :T/1 als Eingabedatei und liest die Zeilen wieder ein.

Verwenden Sie CF bei Dateien, die nicht mehr bearbeitet werden sollen, wird der von EDIT benötigte Speicherplatz verringert.

4.2.9 Schleifen

Bei vielen Befehlen können Sie eine Wiederholung bewirken, indem Sie ihnen eine Zahl (ohne Vorzeichen) voranstellen. Zum Beispiel:

```
24N
```

Befehlsgruppen können in ähnlicher Form mehrmals ausgeführt werden. Zum Beispiel:

```
12(F/hadlich/; E/hadlich/handlich/)
```

Geben Sie als Zahl null (0) ein, wird der Befehl oder die Befehlsgruppe so oft wiederholt, bis EDIT das Ende der Quelldatei erreicht.

4.2.10 Umfassende Operationen

Umfassende Operationen sind Operationen, die automatisch durchgeführt werden, während EDIT die Quelldatei nach unten abarbeitet. Umfassende Operationen werden mit speziellen Befehlen begonnen und beendet. Diese Befehle werden im folgenden Abschnitt beschrieben.

Achtung! Beim Rückwärtsgehen durch die Quelldatei sollten Sie darauf achten, daß alle umfassenden Befehle vorher gelöscht werden. Aktive umfassende Befehle können eine Menge erledigter Arbeit zunichte machen!

4.2.10.1 Festlegen von umfassenden Änderungen

Drei Befehle, GA, GB und GE, stehen für Zeichenketten-Änderungen zur Verfügung. Ihre Syntax lautet wie folgt:

```
GA q/s/t/
GB q/s/t/
GC q/s/t/
```

Diese Befehle führen einen A-, B- oder E-Befehl wie gewohnt an jeder Stelle durch, an der die Zeichenkette »s« in einer neuen aktuellen Zeile erscheint. Sie werden jeweils in der Zeile ausgeführt, die gerade aktuell ist.

G-Befehle bearbeiten ihren ersetzenden Text nicht nochmals. Der folgende Befehl ist deshalb keine Endlos-Schleife, hätte aber keine sichtbaren Auswirkungen in irgendeiner Zeile.

```
GE/rote Rüben/rote Rüben/
```

Das einzige Ergebnis der *Änderung* wäre die Verifizierung der Zeilen mit dem Inhalt »rote Rüben«.

EDIT führt die umfassenden Änderungen in jeder neuen aktuellen Zeile in der Reihenfolge aus, in der Sie sie angegeben haben.

4.2.10.2 Aufheben von umfassenden Änderungen

Der Befehl REWIND bricht alle umfassenden Operationen ab. Mit dem Befehl CG (Cancel Global) können einzelne umfassende Befehle abgebrochen werden.

Wurde eine umfassende Operation mit den Befehlen GA, GB oder GE gestartet, ist der Operation eine Kenn-Nummer zugewiesen, die in die Verifizierungsdatei ausgegeben wird; zum Beispiel G1. Das Argument von CG ist die Nummer der umfassenden Operation, die abgebrochen werden soll. Wird CG ohne Argument ausgeführt, löscht EDIT alle umfassenden Operationen.

4.2.10.3 Aufschub umfassender Änderungen

Einzelne umfassende Änderungen können mit dem Befehl DG (Disable Global) vorübergehend aufgehoben und mit EG (Enable Global) wieder eingesetzt werden. Auch diese Befehle benötigen die Kenn-Nummer der jeweiligen Operation als Argument. Lassen Sie das Argument fort, schaltet EDIT alle umfassenden Operationen an oder aus.

4.2.11 Anzeigen des Programmstatus

Zwei Befehle, die mit SH (SHow) beginnen, geben Informationen über den Status von EDIT an die Verifizierungsdatei aus. Der Befehl SHD (SHow Data) hat folgende Form:

```
SHD
```

Er zeigt gespeicherte oder voreingestellte Informationen, wie zum Beispiel den letzten Suchbegriff. Der Befehl SHG (SHow Globals) hat die Form:

```
SHG
```

Er zeigt die aktuellen umfassenden Befehle, zusammen mit ihrer Kenn-Nummer. Ebenso wird die Anzahl der durchgeführten Änderungen zu jeder umfassenden Operation angegeben.

4.2.12 Beenden eines EDIT-Ablaufs

Der Befehl W (Windup) wird zum *Vorspulen* der restlichen Quelldatei verwendet. Wird das Dateiende erreicht und sind alle Dateien geschlossen und der Puffer-Speicher geleert, wird EDIT dadurch beendet.

Rufen Sie EDIT nur mit dem FROM-Dateinamen auf, so benennt EDIT die zeitweilige Ausgabedatei, die während der Arbeit erzeugt wurde, mit dem gleichen Namen wie dem des Originals. Das heißt, die Ausgabedatei bekommt den Namen der FROM-Datei, während der ursprüngliche Inhalt der FROM-Datei in der Datei :T/EDIT-BACKUP abgelegt wird. Diese

Sicherheitskopie (Backup) der Datei ist natürlich nur bis zu einem erneuten Ablauf von EDIT verfügbar. W ist unzulässig, wenn die Ausgabe momentan nicht auf TO gerichtet ist.

Der Befehl STOP beendet EDIT sofort. Es werden keine weiteren Ein- oder Ausgaben durchgeführt. Die Quelldatei wird nicht überschrieben. Die Eingabe von STOP bietet die Sicherheit, daß in der Eingabedatei keine Änderung durchgeführt werden.

Q unterbricht die Ausführung der aktuellen Befehlsdatei. EDIT liest Befehle normalerweise von der Tastatur ein, jedoch können auch Befehlsdateien mit dem Schlüsselwort WITH oder dem Befehl C angegeben werden. Wurde die Ausführung der aktuellen Befehlsdatei abgebrochen, kehrt EDIT zur vorhergehenden Eingabe-Ebene zurück. Ein Q in der ersten Ebene bewirkt das gleiche wie W.

4.2.13 Verifizieren der aktuellen Zeile

Eine Zeile wird automatisch verifiziert (neu ausgegeben), wenn:

- ein neuer Befehl eingegeben wurde, der Veränderungen an der aktuellen Zeile bewirkte,
- eine Zeile noch nicht ausgegeben wurde, seit sie aktuell ist,
- zu einer veränderten, aber nicht ausgegebenen Zeile zurückgesprungen wird,
- eine Fehlermeldung ausgegeben wird.

In den ersten drei Fällen wird nur verifiziert, wenn die Option V auf EIN gestellt ist. Der folgende Befehl ändert die Einstellung von V.

```
V sw
```

V steht auf EIN, wenn beim Programmstart nicht anders angegeben. Interaktiv wird gearbeitet, wenn Befehle und Verifizierungen mit dem Terminal verbunden sind.

Um die Verifizierung der aktuellen Zeile zu erreichen, verwenden Sie den Befehl ?.

```
?
```

Die Verifizierung wird automatisch ausgeführt, wenn die Option V eingeschaltet ist und der Inhalt der Zeile geändert wurde. Die Verifizierung besteht aus der Zeilennummer oder »++++«, wenn die Zeile nicht zur Quelldatei gehört, und dem Text auf der Zeile.

Eine andere Form der Verifizierung der aktuellen Zeile bietet der Befehl »!«. Er wird eingesetzt, um zusätzlich eine Statuszeile auszugeben. Die Syntax dieses Befehls lautet:

```
!
```

Der Befehl ! verifiziert die aktuelle Zeile mit Zeichenindikatoren. EDIT zeigt zwei Zeilen. Die erste ist die aktuelle Zeile. In der zweiten Zeile zeigt EDIT ein Minus-Zeichen unter all jenen Zeichen, die Großbuchstaben sind und ein Kennzeichen für Steuerzeichen. Alle

anderen Stellen enthalten Leerzeichen. Zum Beispiel zeigt der Befehl ! unter einer Zeile, die einen Tabulatorsprung enthält, an:

```
4.
Diese Zeile enthält einen            Tabulatorsprung.
-     -                T...........-
```

4.2.14 Verschiedenartige Befehle

Dieser Abschnitt beschreibt all jene Befehle, die sich nicht in die zuvor beschriebenen Kategorien einfügen. Er beschreibt, wie das Beendigungszeichen verändert wird, wie nachfolgende Leerzeichen ausgeschaltet, Zeilen umnumeriert und die Quelldatei *zurückgespult* wird.

Um das Zeichen zum Abschluß einer Texteinfügung zu ändern, wird der Befehl Z verwendet. Er hat die folgende Form:

```
Z/s/
```

wobei /s/ für eine Zeichenkette steht. Diese kann bis zu 16 Zeichen lang sein. Groß- und Kleinbuchstaben sind erlaubt. Mit dem Qualifikator PU kann nach dem Zeichen gesucht werden. Das ursprüngliche Zeichen ist »Z«.

Um *nachfolgende Leerzeichen* (am Ende der Zeile) ein- oder auszuschalten, verwenden Sie den Befehl TR (TRailing spaces). Der Befehl TR hat die folgende Form:

```
TR sw
```

wobei sw für EIN (+) oder AUS (–) steht. EDIT unterdrückt gewöhnlich alle am Ende einer Zeile stehenden Leerzeichen. TR+ erlaubt sie in Eingabe- und Ausgabezeilen.

Um Zeilen umzunumerieren, wird der Befehl »=« verwendet. Er hat folgende Syntax:

```
=n
```

wobei n für eine Nummer steht. »=n« ändert die Nummer der aktuellen Zeile zu n. Alle nachfolgenden, auch die nicht zur Quelldatei gehörenden Zeilen werden umnumeriert. In der gleichen Weise werden nach »=« alle vorhergehenden Zeilen in der Ausgabereihe als nicht zur Quelldatei gehörig markiert. Wird EDIT beendet, erhalten alle Zeilen, ob Quellzeile oder nicht, und auch solche Zeilen, die zuvor schon umnumeriert wurden, eine neue Nummer.

Um die Quelldatei zurückzuschreiben, verwenden Sie den Befehl REWIND. Seine Syntax lautet:

```
REWIND
```

Dieser Befehl spult die Eingabedatei so zurück, daß Zeile 1 wieder zur aktuellen wird. Zuerst untersucht EDIT die restliche Quelldatei, um umfassende Operationen auszuführen. Dann

werden alle Zeilen in die Zieldatei geschrieben, diese geschlossen und als neue Quelldatei wieder geöffnet. Die ursprüngliche Quelldatei wird geschlossen und eine temporäre Datei als Zieldatei verwendet. Alle umfassenden Operationen werden beendet. EDIT erkennt diesen Befehl bereits, wenn »REWI« eingetippt wird.

4.2.15 Abbrechen der interaktiven Editierung

Um Befehle abzubrechen, die nach einem String suchen, verwenden Sie <Ctrl>-C. Haben Sie den Suchbegriff falsch eingegeben, bricht <Ctrl>-C die Suche ab. Entsprechend gibt der Befehl T bis zum Ende der Quelldatei aus, wird aber mit <Ctrl>-C abgebrochen.

Nach Eingaben von <Ctrl>-C gibt EDIT folgende Meldung aus:

```
*** BREAK
```

und kehrt in den Eingabe-Modus zurück. Die aktuelle Zeile ist die, die sich im Speicher befand, als Sie <Ctrl>-C betätigt hatten.

4.3 Kurzübersicht

In dieser Liste werden als Abkürzungen verwendet:

Abkürzung	**Beschreibung**
qs	Qualifizierte Zeichenkette
t	Zeichenkette
n	Zeilennummer (oder ».« für aktuelle Zeile bzw. »*« für letzte Zeile)
sw	+ oder – (für »ein« und »aus«)

Befehle in Zeilenfenstern

Befehl	**Funktion**
<	Zeiger nach links setzen.
>	Zeiger nach rechts setzen.
#	Zeichen löschen.
$	Zeichen zu Kleinbuchstaben ändern.
%	Zeichen zu Großbuchstaben ändern.
_	Zeichen löschen und Leerzeichen einfügen.
PA qs	Bewegt Zeiger nach qs.
PB qs	Bewegt Zeiger vor qs.
PR	Bewegt Zeiger zum Zeilenanfang.

Befehle zum Zeilensprung

Befehl	Funktion
M n	Geht zu Zeile n.
M +	Geht zur ersten Zeile im Speicher.
M –	Geht zur letzten Zeile im Speicher.
N	Geht zur nächsten Zeile.
P	Geht zur vorhergehenden Zeile.
REWIND	Liest vorheriges Zielfile als Quellfile ein.

Befehle zum Suchen von Strings

Befehl	Funktion
F qs	Findet String qs.
BF qs	Wie F, sucht zum Anfang des Files.
DF qs	Wie F, löscht alle Zeilen bis zur gesuchten.

Befehle zum Ausgeben von Test

Befehl	Funktion
?	Zeigt aktuelle Zeile.
!	Zeigt Zeile und Statuszeile.
T	Gibt alle Zeilen bis zum Fileende aus.
T n	Gibt n Zeilen aus.
TL n	Gibt n Zeilen mit Zeilennummer aus.
TN	Gibt Zeilen aus, bis alle Zeilen im Puffer geändert wurden.
TP	M- und dann alle verfügbaren Zeilen ausgeben.
V sw	Schaltet Verifizierung ein oder aus.

Operationen in der aktuellen Zeile

Befehl	Funktion
A qs t	Setzt String t hinter qs ein.
AP qs t	Wie A, bewegt aber den Zeiger.
B qs t	Setzt String t vor qs ein.
BP qs t	Wie B, bewegt aber den Zeiger.
CL t	Verbindet aktuelle Zeile, String t und nächste Zeile zu einer Zeile.
D	Löscht die aktuelle Zeile.
DFA qs	Löscht Zeile ab dem Ende von qs.
DFB qs	Löscht Zeile ab dem Anfang von qs.
DTA qs	Löscht vom Anfang der Zeile bis hinter qs.
DTB	Löscht vom Anfang der Zeile bis vor qs.
E qs t	Ersetzt qs durch t.

Befehl	Funktion
EP qs t	Wie E, bewegt aber den Zeiger.
I	Setzt Text von der Tastatur vor der aktuellen Zeile ein.
I t	Übernimmt Text aus File t.
R	Ersetzt Text mit Text vom Terminal.
R t	Ersetzt Text mit Text aus File t.
SA qs	Trennt Zeile hinter qs.
SB qs	Trennt Zeile vor qs.

Umfassende Befehle

Befehl	Funktion
GA qs t	Wirkt wie A, jedoch im gesamten Quellfile.
GB qs t	Wirkt wie B, jedoch im gesamten Quellfile.
GE qs t	Wirkt wie E, jedoch im gesamten Quellfile.
CG n	Löscht umfassenden Befehl Nummer n (oder alle, wenn n nicht angegeben wird).
DG n	Setzt umfassenden Befehl Nummer n außer Kraft (oder alle, wenn n nicht eingegeben wird).
EG n	Aktiviert umfassenden Befehl Nummer n wieder (oder alle, wenn n nicht eingegeben wird).
SHG	Zeigt Informationen über alle umfassenden Befehle.

Ein- und Ausgabe-Befehle

Befehl	Funktion
FROM	Übernimmt Text aus Quelldatei.
FROM t	Übernimmt Text aus File t.
TO	Schreibt Text in Zieldatei.
TO t	Schreibt Text an File t.
CF t	Schließt File t.

Sonstige Befehle

Befehl	**Funktion**
'	Wiederholt vorangehenden A-, B- oder E-Befehl.
= n	Numeriert Datei neu durch, beginnend in der aktuellen Zeile mit n.
C t	Übernimmt Befehle aus File t.
H n	Setzt Stop-Zeiger in Zeile n (wenn n=*, wird Zeiger gelöscht).
Q	Beendet Befehlsebene, wie W, wenn in Ebene 1.
SHD	Zeigt Informationen über den Programmstatus.
STOP	Beendet EDIT unverzüglich.
TR sw	Erlaubt oder verbietet Leerzeichen am Zeilenende.
W	Beendet EDIT und schreibt Puffer in die Zieldatei.
Z t	Definiert t als Zeichen zum Abschluß eingefügten Textes.

AMIGA

DOS-Handbuch

2. Buch

Das Programmierer-Handbuch

Preferences einsetzen

Der Amiga stellt in einem normal breiten CLI-Fenster 60 Zeichen pro Zeile dar. Viele Programmierer jedoch ziehen die 80-Zeichen-Darstellung vor. Diese Einstellung wird mit Programm PREFERENCES auf der Workbench-Diskette verändert. Die aktuellen CLI-Fenster werden bei dieser Einstellung jedoch nicht mit verändert. Jedes bereits geöffnete Fenster bleibt also bei der 60-Zeichen-Darstellung. Um auch sofort mit 80 Zeichen pro Zeile arbeiten zu können, wird aus dem alten Fenster ein neues aufgerufen und das alte gelöscht.

Machen Sie mit:

1. Rufen Sie das Kommando NEWCLI auf.

2. Klicken Sie das alte Fenster an.

3. Tippen Sie ENDCLI in das alte Fenster, um es zu löschen.

Haben Sie PREFERENCES von der Workbench aus aufgerufen, ändert sich das aktuelle CLI nicht. Erst nach dem Reboot sehen Sie die Veränderung, wenn Sie PREFERENCES gesichert haben.

Kapitel 5: Die Programmierung des Amiga

Dieses Kapitel führt den Leser in die Programmierung des Amiga in C oder Assembler unter AmigaDOS ein.

5.1 Einführung

AmigaDOS läßt sich mit dem Amiga, einer SUN-Workstation und dem IBM-PC zur Programmierung des Rechners einsetzen.

Dieses Handbuch setzt voraus, daß Sie schon Erfahrung im Programmieren mit C oder 68000-Assembler haben. Eine Einführung in C finden Sie in dem Buch *Das C-Buch* (tewi-Verlag, Best.-Nr. 80362, München, 1986). Speziell für die Programmierung in C auf dem Amiga gibt es die beiden Werke *Amiga-Programmier-Handbuch* (M&T-Verlag, Best.-Nr. 90491, Haar bei München, 1987) und *Amiga: C in Beispielen* (M&T-Verlag, Best.-Nr. 90539, Haar bei München, 1987). Zur Programmierung in MC68000-Assembler sind einige Bücher auf dem Markt, darunter auch das *Amiga-Assembler-Buch* (M&T-Verlag, Best.-Nr. 90525, Haar bei München, 1987).

5.2 Programmentwicklung für den Amiga

Dieser Abschnitt zeigt die Entwicklung von Programmen für den Amiga. Er beschreibt, was vor Beginn der Arbeit gebraucht wird, den Aufruf von System-Routinen und die Erstellung eines auf dem Amiga ausführbaren Files.

Achtung! Fertigen Sie zuerst eine Kopie Ihrer Systemdiskette an! Eine Anleitung dazu finden Sie am Anfang des Anwenderhandbuchs

5.2.1 Zum Beginn

Um Ihren Amiga zu programmieren, benötigen Sie:

1. Eine Dokumentation zu AmigaDOS und den Systemroutinen, die in Programme eingebunden werden sollen. Dazu gehören zum Beispiel das *AmigaDOS-Anwender-Handbuch*, das *ROM-Kernal-Handbuch* und das *Technische AmigaDOS-Handbuch.*

2. Ein Handbuch zu der Programmiersprache, in der Sie programmieren wollen. Ist das C oder Assembler, finden Sie in diesem Handbuch Hinweise zur Verwendung dieser Sprachen.

3. Include-Files mit den Systemroutinen, die Sie in Ihre Programme einbinden wollen. Alle Include-Files für C enden mit einem ».h«, alle für Assembler mit ».i«. Das jeweilige Include-File wird dann in den Quellcode eingebunden. Um zum Beispiel AmigaDOS in ein Programm in C einzubinden, wird das File DOS.H verwendet.

4. Einen Assembler (zum Beispiel DevPac (M&T-Verlag, Best.-Nr. 51656) oder einen C-Compiler (zum Beispiel Lattice C oder Aztec C), entweder für den Amiga selbst oder für einen der Rechner, die ein Cross-Development (Entwicklung von Programmen für den Amiga auf einem anderen Rechner) ermöglichen.

5. Den Amiga-Linker, für den Amiga selbst oder für einen Cross-Development-Rechner sowie die Amiga-Library-Files und Interface-Routinen.

6. Alle Programme zur Programmübertragung von einem anderen Rechner zum Amiga, falls Sie im Cross-Development programmieren.

5.2.2 Der Aufruf residenter Libraries

Der Aufruf von ROM- und System-Routinen unterscheidet sich in C und Assembler grundlegend. C-Programme rufen lediglich eine Funktion direkt auf. In Assembler wird der jeweilige *library base pointer* (Basisadresse der Library-Routinen) für die aufzurufende Routine in Register A6 geladen und dann zu einem passenden negativen Offset dieses Pointers gesprungen. Diese Offsets sind in der Systembibliothek des Amiga in der Form *_LVOname* gespeichert. Der Aufruf dieser Funktion lautet also *JSR _LVOname(A6),* wenn A6 mit der jeweiligen Library-Basisadresse geladen ist. Diese Basisadressen erhalten Sie durch die Funktion OpenLibrary() der Exec-Bibliothek; die Basisadresse der Exec-Bibliothek wiederum ist in Position 4 abgelegt (der einzigen absoluten Position des Amiga). Diese Position wird auch als AbsExecBase bezeichnet und ist in Amiga.lib so definiert (im *ROM Kernal Manual* finden Sie weitere Informationen über Exec).

Auf diesem Weg können alle im RAM abgelegten Libraries und die AmigaDOS-Library aufgerufen werden. Beachten Sie, daß die AmigaDOS-Library unter der Bezeichnung DOS.LIBRARY aufgerufen werden muß. Als Register für die Basisadresse müssen Sie nicht unbedingt A6 verwenden, jedes andere Register steht Ihnen ebenso zur Verfügung. Die AmigaDOS-Library können Sie übrigens auch durch den Bibliotheks-Aufruf des Linkers in Ihr Programm einbinden. In diesem Fall brauchen Sie nur ein JSR zum Einsprungpunkt zu programmieren, der Linker weiß dann, daß Sie eine Systemroutine einbinden wollen. Mit dem Einladen des Quellcodes in den Speicher öffnet der Rechner dann automatisch die angesprochene Systembibliothek und schließt sie wieder, wenn Ihre Arbeit beendet ist. Die Lade-Routine schreibt die Basisadresse der genutzten AmigaDOS-Systemroutinen in die Einsprung-Punkte von AmigaDOS, um einen korrekten Offset zu erreichen.

5.2.3 Erstellen eines ausführbaren Programmes

Um ein ausführbares Programm auf dem Amiga zu erstellen, gehen Sie am besten die folgenden vier Punkte durch. Jeder Schritt kann auf dem Amiga selbst oder einem zum Cross-Development geeigneten Computer ausgeführt werden.

1. Bringen Sie den Quellcode auf den Entwicklungsrechner. Dazu können Sie diesen mit dem Editor direkt erstellen oder ihn von einem anderen Rechner übertragen. Mit den Befehlen READ und DOWNLOAD können Text- oder Binär-Files zum Amiga übertragen werden.

2. Assemblieren oder kompilieren Sie Ihr Programm.

3. Linken Sie das File mit allen Systemroutinen und dem jeweils benötigten Startup-Code.

4. Laden Sie das fertige Programm in den Amiga und starten Sie einen Probelauf.

5.3 Ausführen eines Programmes vom CLI aus

Ein Programm kann auf zwei Wegen gestartet werden. Erstens kann ein Programm aus einem CLI-Task aufgerufen oder zweitens von der Workbench aus gestartet werden. Dieser Abschnitt beschreibt den ersten Weg.

Der Programmaufruf vom CLI aus gleicht dem eines weniger bedienerfreundlichen Rechners. Dennoch kann er nützlich sein, wenn sich ein Programm noch in der Entwicklungsphase befindet.

5.3.1 Grundsätzliches zur Programmierung in Assembler

Ein Programm wird vom CLI gestartet, indem Sie den Namen des Programmes und eventuelle Argumente eingeben. Die Ein- und Ausgaberichtung kann mit den Befehlen »<«

und »>« definiert werden. Das CLI stellt dem auszuführenden Programm alle diese Informationen zur Verfügung.

Wenn das CLI ein Programm startet, richtet es normalerweise einen Stack von 4000 Byte nur für dieses Programm ein. Die Größe dieses Stacks kann mit dem Befehl STACK verändert werden. Der Stack wird errichtet, bevor das Programm ausgeführt wird. Er ist nicht mit dem Stack des CLI identisch. AmigaDOS schreibt eine Rücksprung-Adresse in diesen Stack, mit der die Kontrolle wieder an das CLI übertragen wird, wenn die Programmausführung beendet ist. Unterhalb der Adresse 4(SP) wird die Größe des Stack in Bytes abgelegt.

In Register A0 wird der Zeiger auf die Argumente, die Ihr Programm benötigt, abgelegt. AmigaDOS legt diese Argumente auf den Stack des CLI, der Zeiger bleibt während des Programmablaufes unverändert. Register D0 enthält die Zeichenzahl der übergebenen Argumente. Mit diesen Werten können Sie die Argumente decodieren. Beachten Sie bitte, daß dieses Register während des Programmablaufes überschrieben werden kann.

Die Standard-Input- und -Output-Routinen sind in den AmigaDOS-Libraries als Input() und Output() abgelegt. Die Aufrufe erlauben alle normalerweise von Anwendern benötigten Input- und Output-Richtungen. Normalerweise wird zur Ein- und Ausgabe das Terminal verwendet. Die Änderung der Ein- oder Ausgaberichtung wird mit den Befehlen »<« und »>« in der Übergabezeile für Argumente definiert. Files, die auf diese Weise geöffnet wurden, sollten nicht innerhalb des Programmes geschlossen werden. Diese Aufgabe übernimmt das CLI für Sie.

5.3.2 Grundsätzliches zur Programmierung in C

Zu Beginn eines C-Programmes sollten Sie immer den Startup-Code einbinden. Der Linker setzt das Programm dadurch automatisch an den Startup Code Entry Point. Dieser Programmteil überprüft die eingegebenen Argumente und schreibt sie nach ARGV, die Anzahl der Elemente wird in ARGC abgelegt. Ebenso werden die AmigaDOS-Libraries geöffnet, die Standard-Input/Output-Files den Files STDIN und STDOUT zugeordnet und dann die Funktion MAIN aufgerufen.

5.3.3 Fehlermeldungen der AmigaDOS-Library-Routinen

Die meisten AmigaDOS-Routinen melden einen Fehler mit dem Return-Code Null. Nur Lese- und Schreibzugriffe melden –1 als Fehlercode. Die Funktion IoErr() liefert eine Integer-Zahl, die den im Anhang des *AmigaDOS-Anwender-Handbuchs* beschriebenen Fehlermeldungen entspricht. Damit erhalten Sie exakte Informationen über einen aufgetretenen Fehler.

5.3.4 Beenden eines Programmes

Ein Assembler-Programm wird am einfachsten mit einem RTS beendet. Dabei sollten Sie natürlich den Stackpointer verwenden, der am Anfang Ihres Programmes gültig war. In diesem Fall sollten Sie einen Return-Code in Register D0 übergeben. Ist kein Fehler aufgetreten, so übergeben Sie Null, anderenfalls eine positive Zahl. Enthält das Register eine anderen Wert außer Null, registriert das CLI einen Fehler. Abhängig von dem mit FAILAT eingestellten Abbruchwert wird ein nicht interaktiv arbeitendes CLI dann abgebrochen (wie zum Beispiel ein EXECUTE-File, das Ihr Programm aufgerufen hat). Ein in C geschriebenes Programm kann einfach durch die Rückkehr aus MAIN beendet werden. Dabei wird zum Startup-Code zurückgekehrt, Register D0 wird gelöscht und ein RTS erzeugt.

Alternativ zu dieser Art, ein Programm zu beenden, kann ein Programm auch die AmigaDOS-Funktion EXIT aufrufen. Sie erhält den Return-Code als Argument. Dadurch wird das Programm abgebrochen, unabhängig davon, welcher Wert im Stackpointer steht.

Wichtig ist in diesem Zusammenhang noch, daß AmigaDOS keine Ressourcen verwaltet. Das muß vom Programmierer übernommen werden. Alle vom Programm geöffneten Files müssen geschlossen, alle Locks geöffnet und eventuell gesperrter Speicher wieder freigegeben werden. Natürlich ist es möglich, Programme zu schreiben, die eine Ressource geöffnet lassen, zum Beispiel einen Programmteil laden, der erst später benutzt werden soll. Dieses Programm sollte aber eine Routine zum Löschen aller offenen Ressourcen enthalten.

5.4 Ausführen eines Programmes von der Workbench aus

Um ein Programm auf der Workbench zu starten, müssen Sie den verschiedenen Möglichkeiten, ein Programm laufen zu lassen, Rechnung tragen. Unter dem CLI läuft ein Programm als Teil des CLI-Prozesses. Die Ein- und Ausgabe und andere Informationen können vom CLI übernommen werden, ebenso die Verwaltung Ihrer Argumente.

Starten Sie ein Programm jedoch auf der Workbench, läuft es als neuer Prozeß zur gleichen Zeit wie die Workbench. Diese lädt das Programm und startet es dann. Sie müssen die erste *Nachricht* an das Programm abwarten, bevor Sie selbst etwas tun können. Diese Nachricht wird nach Ende des Programmes an die Workbench übergeben, um das Programm aus dem Speicher zu löschen.

Für C-Programmierer ist das kein Problem. Diese benutzen nur eine etwas geänderte Startup-Routine. Anders beim Programmieren in Assembler: Dort muß dies alles extra programmiert werden.

Ebenso sollten Sie beachten, daß beim Programmstart unter der Workbench keine Ein- und Ausgabekanäle definiert sind. Stellen Sie deshalb sicher, daß Ihr Programm alle Ein- und Ausgabekanäle selbst öffnet und schließt.

5.5 Cross-Development für den Amiga

Dieser Abschnitt beschreibt das Cross-Development für den Amiga. Dabei wird der Objekt-Code per Download zum Amiga befördert. Speziell beschrieben wird die Übertragung von einem SUN-Mikrosystem und einem Rechner unter MS-DOS. Die Entwicklung mit anderen Systemen wird ebenfalls kurz angesprochen.

5.5.1 Cross-Development auf einer SUN-Workstation

Die zum Cross-Development auf einer SUN zur Verfügung stehenden Programme sind ein Linker, ein Assembler und zwei C-Compiler. Das Format, das Assembler und Linker zur Übergabe von Argumenten verwenden, entspricht dem des Amiga unter dem CLI. Der C-Compiler der Firma Greenhill ist nur für die SUN erhältlich und wird unten beschrieben.

Der Compiler wird METACC genannt. Er akzeptiert mehrere Filetypen: Ein Filename mit der Endung ».c« enthält ein C-Quell-Programm. Der Compiler kompiliert dieses Programm und schreibt das Ergebnis mit gleichem Namen, jedoch mit der Endung ».obj« (für Objektcode) in das gleiche Directory. Die Endung ».asm« identifiziert ein File als Assembler-Quellcode. Dieses File wird mit dem Assemblerdurchlauf zu einem ».obj«-File im gleichen Directory.

Metacc unterstützt einige wichtige Optionen in folgendem Format:

```
metacc [<opt1> [,<opt2> [,...<optn>]]] [<file>[,...<filen>]]
```

Zur Verfügung stehen folgende Optionen:

```
 -c -g -go -w -p -pg -O[<optflags>]
-fsingle -s -E -C -X70 -o <output> -D <name>=def>
 -U <name> -I <dir> -B <string> -t[p012]
```

Diese Optionen bewirken:

-c kompiliert nur das Programm, lädt keine Files nach, produziert nur ein Objektfile, selbst wenn ein Programm kompiliert wird.

-g erzeugt eine zusätzliche Symboltabelle für den Debugger DBX und setzt das Flag -lg auf ld.

-go erzeugt eine zusätzliche Symboltabelle für den älteren Debugger ADB und setzt das lg-Flag auf ld.

-w unterdrückt alle Fehlermeldungen.

-p erzeugt Code zum Zählen der einzelnen Aufrufe von Routinen. Wird das File geladen, wird die normale Startup-Routine von einer ersetzt, die aus dem Monitor aufgerufen wird, und nutzt andere als die normalen

C-Bibliotheken. Setzen Sie das Programm PROF ein, um ein Ausführungsprofil zu erstellen.

-pg erzeugt ähnlich -p einen Profile-Code. Das Programm ruft jedoch einen Run-Time-Mechanismus auf, der ein GMON.OUT-File erstellt, das genauere Informationen über das kompilierte File enthält. Setzen Sie das Programm GPROF ein, um ein Ausführungsprofil zu erstellen.

-O[<optflags>] setzt den Objektcode-Optimizer ein, um den erzeugten Code zu verbessern. Wenn <optflags> erscheinen, werden diese in der Befehlszeile eingesetzt, um den Optimizer zu starten. -O kann zum Setzen der Optionflags verwendet werden.

-fsingle schaltet auf Arithmetik mit einfacher Genauigkeit. (Doppelte Genauigkeit ist Standardeinstellung.)
Fließkomma-Zahlen werden weiter mit doppelter Genauigkeit berechnet. Und von Funktionen zurückgelieferte Zahlen sind ebenfalls doppelt genau.

Achtung! Viele Programme laufen wesentlich schneller, wenn sie mit der Option fsingle kompiliert sind. Beachten Sie aber, daß ungenaue Werte ein Programm oft sinnlos machen.

-S kompiliert die angegebenen C-Programme und erzeugt zusätzlich ein Assemblerfile. Beide enden mit ».obj«.

-E startet nur den C-Preprozessor mit den angegebenen Programmen. Der Output geht an das normale Output-Device.

-C weist den C-Preprozessor an, Kommentare nicht zu löschen.

-X70 erzeugt arithmetischen Code im Fließkomma-Format des Amiga. Dieser Code ist kompatibel zu den Fließkomma-Routinen der mathematischen ROM-Bibliothek des Amiga.

-o <output> nennt das Ausgabefile <output>. Wenn Sie diese Option einsetzen, wird das File a.out nicht gelöscht.

-D<name=def> definiert <name> als Preprozessornamen, als wäre er mit einem #define definiert worden. Ist kein <def> angegeben, wird der Name als 1 definiert.

-U<name> löscht jede Definition von <name>.

-I<dir> sucht zuerst im Directory, das mit <file> angegeben wird, nach einem #include- File, dessen Name nicht mit »/« beginnt. Die Suche geht dann im Directory weiter, das mit <dir> übergeben wurde, und zum Schluß im Directory /usr/include.

-B<string> sucht in den mit <string> übergebenen Files mit den Endungen CPP, CCOM und C2 nach alternativen Compiler-Passes.

-t[p012] verwendet nur die in einer -B-Option angegebenen Compiler-Passes. Wurde -B nicht verwendet, wird /USR/NEW/ als <string> verwendet.

Die Buchstaben-Zahlen-Kombinationen nach -t haben folgende Bedeutung:

p = cpp – Der C-Preprozessor

0 = metacom – beide Passes des Compilers, nicht jedoch der Optimizer.

1 = Wird ignoriert. Diese Option steht für die zweite Phase eines Zwei-Pass-Compilers. Bei einer SUN enthält ccom jedoch beide Passes.

2 = c2 – Der Objektcode-Optimizer.

METACC faßt alle anderen Argumente als Programme und Bibliotheken auf. Wenn Sie die Optionen -c, -S, oder -E nicht angeben, lädt METACC diese Programme oder Bibliotheken und alle angegebenen kompilierten oder assemblierten Files und erzeugt daraus ein Programm A.OUT. Um diesen Namen zu ändern, können Sie die Option -o<name> verwenden.

Tabelle 5.1 zeigt alle File-Endungen und deren Bedeutung bei Verwendung von METACC:

Beschreibung	Filename
C-Quellcode	file.c
Assembler-Quellcode	file.asm
Objektfile	file.obj
Bibliothek von Objektfiles	file.lib
Ausführbares Objektfile	a.out
Temporäre Datei	/tmp/ctm
Preprozessor	/lib/cpp
Compiler	/lib/ccom
Optionaler Optimizer	/lib/c2
Runtime-Startoff-Code	/lib/crt0.o
Startoff-Code zum Profiling	/lib/mcrt0.o
Startoff-Code zum GPROF-Profiling	/usr/lib/gcrt0.o
Standard-Bibliothek	/lib/lib.c
Profiling-Bibliothek	/usr/lib/libc_p.a
Standard Directory für includes	/usr/include
Von PROF erzeugte Reportfiles	mon.out
Von GPROF erzeugte Reportfiles	gmon.out

***Tabelle 5.1:** Spezielle metacc-Files*

Die vom Linker der SUN erzeugten Files können auf drei Wegen zum Amiga übertragen werden. Der erste und einfachste setzt ein Billboard voraus, der zweite eine parallele Schnittstelle, der dritte eine serielle Schnittstelle.

Haben Sie die *Billboard* genannte Hardware, können Sie Ihr gelinktes File per Download zum Amiga übertragen. (Der Dateiname sollte mit .ld enden):

1. Starten Sie das Programm BINLOAD auf der SUN

```
binload -p &
```

Das braucht (bei mehreren Übertragungen) nur einmal zu geschehen.

2. Geben Sie dann auf dem Amiga in einem CLI-Fenster ein:

```
DOWNLOAD <sun filename> <amiga filename>
```

3. Starten Sie das Programm:

```
<amiga filename>
```

Das könnte zum Beispiel wie folgt aussehen:

Auf der SUN:

```
binload -p&
```

Auf dem Amiga:

```
DOWNLOAD test1.ld test
test
```

Falls sich das File, das zu übertragen ist, nicht im aktuellen Directory der SUN befindet, muß der PATH, der zu dem File führt, angegeben werden. BINLOAD arbeitet auf der SUN immer relativ zum aktuellen Directory. Also wäre der richtige WEG zum File TEST1.LD in den Directories: /USR/COMMODORE/AMIGA/V24/BEISPIELE/DOS/TEST1.LD auf dem Amiga:

```
DOWNLOAD /usr/commodore/amiga/V24/beispiele/dos/test1.ld test
```

Um binload abzubrechen, müssen Sie sein PID mit PS herausfinden und dann KILL verwenden. Beachten Sie bitte, daß binload nach diesem Soft-Reset eine Meldung in den Standard Output Stream schreibt. In der Regel also in das Fenster, aus dem es gestartet wurde. Ist die Übertragung gestört, drücken Sie <Ctrl>-C beim Amiga, um DOWNLOAD abzubrechen. (In Abschnitt 3.2 finden Sie weitere Informationen zu den Abbruch-Flags des Amiga.)

Haben Sie kein Billboard, können Sie Files mit der parallelen Schnittstelle übertragen. Dazu folgen Sie bitte den folgenden Schritten.

1. Senden Sie die ASCII-Files über die parallele Schnittstelle zum Amiga. Tippen Sie dazu:

```
SEND demo.ld
```

Weisen Sie SEND kein Argument zu, wird STDIN verwendet. Das voreingestellt Output-Device ist /DEV/LP0, also das richtige. Um die Ausgaberichtung zu ändern, verwenden Sie die Option -o.

2. Auf dem Amiga geben Sie ein:

```
READ demo
```

READ liest Zeichen von der parallelen Schnittstelle und schreibt sie in das File DEMO.

3. Nach der Übertragung können Sie das Programm auf dem Amiga starten:

```
demo
```

Um Files über die serielle Schnittstelle zu übertragen, ist folgendes nötig:

1. Konvertieren Sie das binäre File zu einem ASCII-Hex-File mit der Endung Q, indem Sie (auf der SUN) eingeben:

```
CONVERT <demo.ld>demo.dl
```

Das File zu diesem Befehl finden Sie in INCLUDE MAKEFILE, MAKEAMIGA. (Im *Technischen AmigaDOS-Handbuch*, Kapitel 11, finden Sie weitere Informationen über die binären Files des Amiga.)

2. Tippen Sie nun auf der SUN:

```
TIP amiga
```

3. Geben Sie danach auf dem Amiga ein:

```
READ demo SERIAL
```

4. Geben Sie schließlich auf der SUN ein:

```
~> demo.dl
```

5. Nun können Sie das Programm auf dem Amiga starten:

```
demo
```

Achtung! Bei der SUN hängt sich die serielle Schnittstelle oft grundlos auf. Dann hilft nur ein Reboot.

Hat sich die Schnittstelle aufgehängt und der Reboot ist abgeschlossen, geben Sie (auf der SUN) ein:

```
TIP
```

und dann

```
Q und <Return>
```

Das beendet auf dem Amiga den Lesevorgang ordnungsgemäß.

5.5.2 Cross-Development unter MS-DOS

Zum Cross-Development auf einem MS-DOS-Rechner brauchen Sie die im Directory \V25\bin untergebrachten Programme. Dort finden Sie einen C-Compiler, einen Assembler, einen Linker und diverse Befehle zur Dateiübertragung. Für diese Befehle gilt dieselbe Syntax auf dem MS-DOS-Rechner und auf dem Amiga.

Zu Download wird der serielle Port (AUX:) benutzt. Hier die Befehle:

1. Geben Sie auf dem Amiga ein:

```
READ file SERIAL
```

2. Auf dem MS-DOS-Rechner:

```
CONVERT <file.ld> AUX:
```

3. Nach der Übertragung kann das Programm auf dem Amiga gestartet werden:

```
file
```

5.5.3 Cross-Development mit anderen Systemen

Zum Cross-Development brauchen Sie einen Cross-Compiler oder Assembler und die Adressen aller Einsprungpunkte. Weiterhin benötigen Sie den Linker ALINK des Amiga, der auf Ihrem Rechner oder dem Amiga laufen muß. Zu guter Letzt benötigen Sie ein Programm für Ihren Rechner, das Binärfiles in ASCII-Hex-Files umwandelt, die mit einem Q enden. (Nur so können Daten mit READ vom Amiga übernommen werden.)

Nun können Sie Ihr Binärfile (über die eine serielle oder parallele Schnittstelle) zum Amiga übertragen, wie es in Abschnitt 5.5.2 beschrieben wurde. Haben Sie das File bereits auf Ihrem MS-DOS-Rechner gelinkt, können Sie komplette ladbare Files übertragen. Andernfalls wird das binäre Objektfile in eine linkfähigen Form übertragen und erst auf dem Amiga gelinkt.

Kapitel 6: Programmieren mit AmigaDOS

Dieses Kapitel beschreibt die Funktionen, die in der Systembibliothek von AmigaDOS enthalten sind.

Dabei sollen Ihnen die Erklärungen zur Syntax ebenso bei der Programmierarbeit helfen wie die komplette Beschreibung aller Funktionen und die Kurzübersicht am Ende des Kapitels.

6.1 Syntax-Vereinbarungen

Die Syntax wird in diesem Kapitel anhand der Funktionsaufrufe für C erklärt. Dabei werden für Assemblerprogrammierer die zugehörigen Register angegeben.

6.1.1 Register-Werte

Die Buchstaben-Zahlen-Kombinationen (D0....Dn) repräsentieren Register. Der Text links des Gleichheitszeichens repräsentiert das Ergebnis einer Funktion. Text rechts des Gleichheitszeichens repräsentiert eine Funktion und ihre Argumente, der Text in Klammern ist dabei die Liste der Argumente. Ein Register (zum Beispiel D0) unter einem Argument oder Funktionsergebnis stellt dar, welcher Wert in welchem Register enthalten ist. Beachten Sie bitte, daß nicht alle Funktionen ein Ergebnis ausgeben.

6.1.2 Schreibweise

Die Schreibweise (Groß- oder Kleinbuchstaben) der Befehlsnamen ist wichtig! Achten sie besonders bei Worten wie »FileInfoBlock« auf die korrekte Schreibweise. Jeder Teil eines zusammengesetzten Befehlsnamens wird normalerweise groß geschrieben.

6.1.3 Wahrheitswerte als Ergebnisse

Der Wert –1 entspricht »Wahr« oder »Erfolgreich« – 0 entspricht »Falsch« oder »Fehlerhaft«.

6.1.4 Werte

Alle Werte werden als Langwort übergeben (4 Byte oder 32 Bit). Als Strings bezeichnen wir 32-Bit-Pointer, die auf eine mit Null endende Zeichenkette zeigen (C-Format).

6.1.5 Format, Argument und Ergebnis

Unter *Argument* und *Ergebnis* finden Sie weitere Details zur Syntax von Befehlen, die mit *Format* beschrieben sind. *Ergebnis* beschreibt die Ausgabe der Funktion (steht links vom Gleichheitszeichen). Argument beschreibt die von der Funktion zur Bearbeitung benötigten Werte (entsprechend der Liste in Klammern). Folgendes Beispiel soll die Syntax erklären:

Format: Ergebnis = Funktion(argument)
Register Register
Beispiel: lock = CreateDir(name)
D0 D1

6.2 Die AmigaDOS-Routinen

Dieser Abschnitt beschreibt die vollständige Syntax aller in der Systembibliothek von AmigaDOS enthaltenen Funktionen. Sie sind alphabetisch nach folgenden Gruppen eingeteilt: File-Handling, Prozeß-Handling und Funktionen zum Laden von Files. Nach jedem Funktionsnamen ist eine kurze Beschreibung der Aufgabe der jeweiligen Funktion abgedruckt. Dahinter finden Sie eine Erklärung zum Format und zu den Registern sowie eine genauere Beschreibung der Funktion und der Syntax mit allen Argumenten und Ergebnissen. Zur Anwendung dieser Funktionen muß das File AMIGA.LIB gelinkt werden.

6.2.1 File-Handling

Close

Aufgabe: Ein Input- oder Outputfile schließen.
Format: Close(file)
D1
Argument: file – File-Handle
Beschreibung: Das File-Handle »file« beschreibt das zu schließende File. Dieses File-Handle ist das Ergebnis der Funktion Open. Die File-Handles aller vom Programm geöffne-

ten Files müssen zwischengespeichert und beim Close des Files angegeben werden. Sie sollten innerhalb eines Programmes keine Files schließen, die außerhalb des Programmes geöffnet wurden.

CreateDir

Aufgabe: Ein neues Directory erzeugen.
Format: lock = CreateDir(name)
D0 D1
Argument: Name – String
Ergebnis: lock – Pointer als Schlüssel
Beschreibung: CreateDir erzeugt ein neues Directory mit dem angegebenen Namen, sofern dies möglich ist. Falls nicht, wird ein Fehler gemeldet. Bedenken Sie bitte, daß AmigaDOS Directories nur in Devices errichten kann, die dazu geeignet sind, zum Beispiel auf Disketten.

Eine Null als Rückmelde-Code zeigt einen Fehler an, zum Beispiel die Diskette war schreibgeschützt. In diesem Fall sollten Sie die Funktion IoErr() aufrufen, andernfalls erhalten Sie einen *shared-read-lock* für das neue Directory.

CurrentDir

Aufgabe: Macht ein Directory zum aktuellen Directory.
Format: oldLock = CurrentDir (lock)
D0 D 1
Argument: lock – Pointer als Schlüssel
Ergebnis: oldLock – Pointer als Schlüssel
Beschreibung: CurrentDir macht ein Directory, dessen Lock Sie übergeben, zum aktuellen Directory. (Siehe auch LOCK.) Als Ergebnis wird der Lock des alten aktuellen Directory übergeben. Das Ergebnis Null ist hier keine Fehlermeldung, sondern zeigt an, daß das aktuelle Directory das Ursprungs-Directory der Start-Diskette ist.

DeleteFile

Aufgabe: Ein File oder Directory löschen.
Format: erfolg = DeleteFile (name)
D 0 D 1
Argument: Name-String
Ergebnis: erfolg – Boolean
Beschreibung: DeleteFile versucht, das mit »name« benannte File oder Directory zu löschen. Gelingt das nicht, wird ein Fehler gemeldet. Beachten Sie bitte, daß aus einem Directory alle Files gelöscht werden müssen, bevor das Directory selbst gelöscht werden kann.

DupLock

Aufgabe: Einen Lock duplizieren.
Format: neuLock = DupLock(lock)
D0 D1
Argument: lock – Pointer zum Schlüssel
Ergebnis: neuLock – Pointer zum Schlüssel
Beschreibung: DupLock erzeugt eine Kopie eines shared-read-lock und liefert diese zurück. Ein write-lock kann nicht dupliziert werden! Weitere Informationen finden Sie unter LOCK.

Examine

Aufgabe: Informationen zu einem Directory oder File erfragen.
Format: erfolg = Examine(lock, FileInfoBlock)
D0 D1 D2
Argument: lock – Pointer zum Schlüssel
FileInfoBlock – Pointer zum File Info Block
Ergebnis: erfolg – Boolean
Beschreibung: Examine schreibt Informationen über das durch den Lock identifizierte File oder Directory in den angegebenen FileInfoBlock. Dieser Block enthält dann den Namen, die Größe, das Datum der Erstellung und den Typ des Files oder Directory.

FileInfoBlock muß als Langwort angegeben werden. In C wird dies mit der Funktion Allocmem erreicht. (Im *ROM Kernal Manual* finden Sie weitere Informationen zum exec-Aufruf von Allocmem.)

ExNext

Aufgabe: Den nächsten Directory-Eintrag untersuchen.
Format: erfolg = ExNext(lock, FileInfoBlock)
D0 D1 D2
Argument: lock – Pointer zum Schlüssel
FileInfoBlock – Pointer zum FileInfoBlock
Ergebnis: erfolg – Boolean
Beschreibung: Diese Routine wird normalerweise verwendet, um die Daten aller Einträge eines Directory in einen vorher mit Examine erstellten FileInfoBlock zu schreiben. ExNext modifiziert diesen Block so, daß weitere Aufrufe von ExNext Informationen zum jeweils nächsten Eintrag liefern, bis alle Directory-Einträge abgearbeitet sind.

ExNext meldet einen Return-Code von Null, wenn ein Fehler aufgetreten ist. Ein Fehler ist jedoch auch das Erreichen des letzten Directory-Eintrags. In IoErr() wird ein exakter Fehlercode eingetragen, mit dem ein *echter* Fehler erkannt werden kann. Die Fehlermeldung des letzten Eintrages lautet dort: ERROR_NO_MORE_ENTRIES.

Mit den folgenden Schritten können Sie ein ganzes Directory untersuchen:

1. Rufen Sie Examine auf, um einen FileInfoBlock für das zu bearbeitende Directory zu erstellen.
2. Übergeben Sie ExNext Lock und FileInfoBlock-Pointer der vorhergehenden Funktion Examine.
3. Rufen Sie ExNext auf, bis IoErr() den Fehler ERROR_NO_MORE_ENTRIES meldet.
4. Im Typ-Feld des FileInfoBlock können Sie feststellen, ob es sich bei dem untersuchten Objekt um ein File oder ein Directory handelt und ob es sich lohnt, ExNext aufzurufen.

Das Typ-Feld des FileInfoBlock kann einen negativen oder einen positiven Wert annehmen. Ist er negativ, handelt es sich bei dem Objekt um ein File, ist er positiv, handelt es sich um ein Directory.

Info

Aufgabe: Liest Informationen über die Diskette.
Format: erfolg = Info(lock, Info_Data)
D0 D1 D2
Argument: lock – Pointer zum Lock
Info_Data – Pointer zu einem Info_Data-Speicherbereich
Ergebnis: erfolg – Boolean
Beschreibung: Info meldet Informationen über alle in Gebrauch befindlichen Disketten. »lock« kann auf die Diskette oder ein beliebiges File darauf verweisen. In den Info_Data-Speicherbereich werden Informationen über die Diskettengröße in Kbyte, die Anzahl der freien und belegten Blocks und die Anzahl der Soft-Errors geschrieben. Info_Data muß übrigens bei einer durch 4 teilbaren Speicheradresse liegen.

Input

Format: file = Input()
D0
Ergebnis: file – File-Handle
Beschreibung: Input liefert das File-Handle für die Standardeingabedatei beim Start. (Das initiale Output-File wird mit OUTPUT festgestellt.)

IoErr

Aufgabe: Zusätzliche Fehlerinformationen vom System erfragen.
Format: fehler = IoErr()
D0
Ergebnis: fehler – Integer
Beschreibung: I/O-Routinen melden als Return-Code Null, wenn ein Fehler aufgetreten ist. Mit IoErr() erhalten Sie mehr Informationen über diesen Fehler. Einige Routinen (zum Beispiel DeviceProc) nutzten IoErr auch, um ein weiteres Ergebnis auszugeben.

IsInteractive

Aufgabe: Feststellen, ob ein File ein virtuelles Terminal (Tastatur und Fenster) ist.
Format: bool = IsInteractive(file)
D0 D1
Argument: file – File-Handle
Ergebnis: bool – Boolean
Beschreibung: Das Ergebnis der Funktion zeigt an, ob ein File ein virtuelles Terminal (Tastatur und Fenster) ist, zum Beispiel ein interaktiv ablaufender CLI-Task.

Lock

Aufgabe: ein File oder Directory für bestimmte Zugriffe sperren.
Format: lock = Lock(name, zugriffsmodus)
D0 D1 D2
Argument: name – String
zugriffsmodus – Integer
Ergebnis: lock – Pointer zu einem Lock
Beschreibung: Lock liefert einen *Lock* auf das genannte File zurück. Ein *Lock* ist gleichzeitig eine eindeutige Identifizierung eines Files und ein Verbot für andere Prozesse, bestimmte Operationen (zum Beispiel Schreib-Operationen) mit dieser Datei durchzuführen. Ist der Zugriffsmodus ACCESS_READ, erhalten Sie einen *shared-read-lock.* Der Lesezugriff auf die Datei ist dann von mehreren Tasks aus möglich. Ist der Modus ACCESS_WRITE, wird ein *write-lock* zurückgegeben. Dann ist ein Schreibzugriff auf die Datei nur aus diesem Programm möglich. Tritt ein Fehler auf, wird eine Null gemeldet.

Um festzustellen, ob ein gesuchtes File existiert, ist es günstiger, Lock anstelle von Open aufzurufen, da der damit verbundene Aufwand geringer ist. Zum Öffnen eines gefundenen Files jedoch muß dann OPEN verwendet werden.

Open

Aufgabe: Ein File zur Ein- oder Ausgabe öffnen.
Format: file = Open(name, zugriffsmodus)
D0 D1 D2
Argument: name – String
zugriffsmodus – Integer
Ergebnis: file – File-Handle
Beschreibung: Open öffnet das File »file« und liefert einen File-Handle darauf zurück. Ist der Zugriffsmodus MODE_OLDFILE (=1005), wird ein bestehendes File zum Lesen oder Schreiben geöffnet. Ein neues File wird zum Schreiben geöffnet, wenn der Modus MODE_NEWFILE (=1006) verwendet wird. »name« ist dann der Name des neuen Files, eventuell mit Angabe eines Device-Namens wie df1:, einem Fenster CON: oder RAW: oder dem aktuellen Fenster (*).

Weitere Informationen zu den Devices finden Sie im Kapitel 1 des *AmigaDOS-Anwender-Handbuches.*

Kann das File nicht geöffnet werden, wird Null als Return-Code übergeben. In diesem Fall finden Sie in IoErr() eine zweite, eindeutige Fehlermeldung.

Um die Existenz eines Files zu testen, lesen Sie bitte unter LOCK nach.

Output

Format: file = Output( )
D0
Ergebnis: file – File-Handle
Beschreibung: Output liefert den File-Handle des initialen Output-Files. (Das initiale Input-File wird mit INPUT festgestellt.)

ParentDir

Aufgabe: Liefert das Directory, in dem ein File oder Directory liegt.
Format: Lock = ParentDir(lock)
D0 D1
Argument: lock – Pointer zu einem Schlüssel
Ergebnis: lock – Pointer zu einem Schlüssel
Beschreibung: Diese Funktion liefert den Lock zu dem Ursprungsdirectory des genannten Files oder Directory.

Beachten Sie bitte, daß diese Funktion den Return-Code Null liefert, wenn das gesuchte Ursprungs-Directory das Ursprungs-Directory des File-Systems ist.

Read

Aufgabe: Daten aus einem File lesen.
Format: aktuellelänge = Read(file, puffer, größe)
D0 D1 D2 D3
Argument: file – File-Handle
puffer – Pointer auf Puffer
größe – Integer
Ergebnis: aktuellelänge – Integer
Beschreibung: Mit einer Kombination aus READ und WRITE können Daten kopiert werden. Read liest Daten aus einem geöffneten File in den mit »puffer« und »größe« errichteten Pufferspeicher. Dabei werden Zeichen gelesen, bis der Puffer ganz gefüllt ist, oder das File-Ende erreicht wird. Sie sollten immer überprüfen, ob der mit »größe« übergebene Wert auch die tatsächliche Größe des Puffers ist. Read meldet zurück, wenn weniger Bytes als angegeben gelesen werden konnten.

Der übergebene Wert zeigt die Anzahl der tatsächlich gelesenen Bytes an. Ist »aktuellelänge« größer Null, entspricht dieser Wert der Anzahl der gelesenen Bytes. Wird Null übergeben, wurde das Ende des Files erreicht. Fehler werden mit dem Return-Code –1 angezeigt. Liest Read von der Tastatur, werden Daten erst dann übergeben, wenn ein <Return> eingegeben wurde oder der Puffer voll ist.

Nach Aufruf von READ kann auch der Wert von IoErr() verändert sein. IoErr() stellt dann weitere Informationen über einen aufgetretenen Fehler zur Verfügung (zum Beispiel aktuellelänge = –1).

Rename

Aufgabe: Ein Directory oder File umbenennen.
Format: erfolg = Rename(altername, neuername)
D0 D1 D2
Argument: altername – String
neuername – String
Ergebnis: erfolg – Boolean
Beschreibung: Rename versucht, dem mit »altername« bezeichneten Directory oder File den Namen »neuername« zuzuweisen. Existiert bereits ein Directory oder File mit dem Namen »neuername«, wird Rename abgebrochen und ein Fehler gemeldet.

»altername« und »neuername« können zusätzlich auch Namen von verschiedenen Directories beinhalten. In diesem Fall wird das File aus dem alten Directory entfernt und mit neuem Namen in das neue Directory kopiert. Das Ziel-Directory muß zu dieser Aktion bereits bestehen.

Beachten Sie bitte, daß Files mit Rename nicht von einer Diskette auf eine andere kopiert werden können.

Seek

Aufgabe: Zu einer bestimmten Position innerhalb eines Files springen.
Format: alteposition = Seek(file, position, modus)
D0 D1 D2 D3
Argument: file – File-Handle
position – Integer
modus – integer
Ergebnis: alteposition – Integer
Beschreibung: Seek setzt den Schreib-Lese-Zeiger von »file« auf die Position »position«. Ab dieser Position wird dann gelesen oder geschrieben (siehe OPEN und READ). Wurde die Funktion fehlerfrei ausgeführt, liefert der Return-Code die »alteposition« im File. Ansonsten wird –1 als Return-Code geliefert. Mit IoErr() erhalten Sie weitere Informationen über den aufgetauchten Fehler.

»modus« kann die Werte OFFSET_BEGINNING (1), OFFSET_CURRENT (0) oder OFFSET_END (–1) annehmen. Damit wird die Startposition innerhalb des Files festgelegt. Zum Beispiel 20 ab Current ist die Position 20 Byte vorwärts ab aktueller Cursor-Position, –20 ab End ist 20 Byte vor dem Ende des Files und so weiter. Um zum Ende des Files zu gelangen, geben Sie als Position 0 und als Modus –1 ein.

Mit Seek können Sie leicht Informationen an ein File anhängen. Gehen Sie zum Ende des Files, wie oben beschrieben, und beginnen Sie zu schreiben. Über das Ende eines Files hinaus kann Seek nicht positionieren.

SetComment

Aufgabe: Einen Kommentar an ein File oder Directory anhängen.
Format: erfolg = SetComment(name, kommentar)
D0 D1 D2
Argument: name – Filename
kommentar – Pointer auf Kommentar-String
Ergebnis: erfolg – Boolean
Beschreibung: SetComment hängt einen Kommentar an ein File oder Directory an. »kommentar« ist der Pointer zu einem String von bis zu 80 Zeichen Länge.

SetProtection

Aufgabe: Die Protect-Flags eines Directory oder Files setzen.
Format: erfolg = SetProtection(name, maske)
D0 D1 D2
Argument: name – Filename
maske – Maske zum Setzen der Flags (s.u.)
Ergebnis: erfolg – Boolean
Beschreibung: SetProtection setzt die Schutzattribute eines Files oder Directory. Die vier niederwertigen Bits der Maske und ihre Funktionen (wenn sie gleich 1 sind) lauten:

Bit 3: Lesen nicht erlaubt.
Bit 2: Schreiben nicht erlaubt.
Bit 1: Ausführung (Execute) nicht erlaubt.
Bit 0: Löschen nicht erlaubt.
Bits 31 bis 4 werden nicht benutzt.

Bei der vorliegenden Version 1.2 von AmigaDOS wird Bit 1 ignoriert. Execute kann also auch ausgeführt werden, wenn Bit 1 gesetzt ist. Der bessere Weg, die Protect-Flags zu setzen, ist es, die neuen Definitionen aus INCLUDE/LIBRARIES/DOS.H zu nutzen.

UnLock

Aufgabe: Aufheben des Locks von einem Directory oder File.
Format: UnLock (lock)
D1
Argument: lock – Pointer zu einem Lock.
Beschreibung: UnLock gibt den Lock, der mit den Funktionen Lock, DupLock oder CreateDir angelegt wurde, frei. Dadurch kann ein neuer Lock (auch ein write-lock) zu diesem Directory oder File erzeugt werden.

WaitForChar

Aufgabe: Erkennen, ob innerhalb der angegebenen Zeit ein Zeichen eingegeben wurde.
Format: bool = WaitForChar(file, zeit)
D0 D1 D2
Argument: file – File-Handle
zeit – Integer
Ergebnis: bool – Boolean
Beschreibung: Konnte innerhalb der mit »zeit« eingegebenen Zeit ein Zeichen aus dem File »file« gelesen werden, wird als Return-Code –1 (wahr) übergeben. Anderenfalls ist der Return-Code 0 (falsch). Wird ein Zeichen erkannt, kann es mit Read gelesen werden.

Beachten Sie bitte, daß WaitForChar nur zulässig (und sinnvoll) ist, wenn die Datei ein virtuelles Terminal ist. »zeit« muß in Mikrosekunden angegeben werden.

Write

Aufgabe: Daten in ein File schreiben.
Format: geschriebenezeichen = Write(file, puffer, größe)
D0 D1 D2 D3
Argument: file – File-Handle
puffer – Pointer zum Puffer
größe – Integer
Ergebnis: geschriebenezeichen – Integer
Beschreibung: Mit einer Kombination aus Read und Write können Daten kopiert werden. Write schreibt Datenbytes in das offene File »file«. »größe« definiert die Anzahl an zu schreibenden Bytes, »puffer« zeigt auf einen Pufferspeicher, der die zu schreibenden Daten enthält.

Write meldet als Return-Code die Anzahl an geschriebenen Bytes zurück, wenn kein Fehler aufgetreten ist und eine »größe« angegeben wurde, die größer als Null ist. Bei einem Fehler ist der Return-Code »–1«. Bei diesem Befehl sollten Sie immer prüfen, ob die Aktion fehlerfrei vonstatten ging. Sie verlieren sonst möglicherweise wichtige Daten.

6.2.2 Prozeß-Handling

CreateProc

Aufgabe: Einen neuen Prozeß starten.
Format: prozeß = CreateProc(name, pri, segment, stack)
D0 D1 D2 D3 D4
Argument: name – String
pri – Integer
segment – Pointer auf Segment
stack – Integer
Ergebnis: prozeß – Prozeß-Kennung
Beschreibung: CreateProc startet einen Prozeß mit Namen »name«. Dazu reserviert der Amiga einen Teil des freien Speichers zur Aufnahme des Programm-Codes und initialisiert ihn.

Dann werden die Segmente, die in der Liste »segment« angegeben wurden, eingeladen. Sie enthalten das eigentliche Programm. CreateProc startet den Prozeß dann durch einen Sprung zum Beginn des ersten Segments.

»stack« gibt die Stackgröße für den neuen Prozeß an. »pri« definiert die Priorität, die dieser Prozeß gegenüber anderen hat. Als Ergebnis wird die Prozeß-Kennung ausgegeben, falls kein Fehler aufgetreten ist. Wurde ein Fehler gefunden, ist der Return-Code null.

DateStamp

Aufgabe: Liefert Datum und Zeit im internen Format von AmigaDOS.
Format: v = DateStamp(v)
Argument: v – Pointer
Beschreibung: DateStamp benötigt einen Pointer auf drei Langworte, in denen die aktuelle Zeit abgelegt wird. Das erste Element ist die Zahl für den Tag. Das zweite Element ist die Anzahl vergangener Sekunden des aktuellen Tages, und das dritte Element gibt die Anzahl vergangener *Ticks* in der aktuellen Sekunde an. Eine Sekunde besteht aus 50 *Ticks*. DateStamp errechnet den Tag und die Minuten folgerichtig. Alle drei Elemente sind null, wenn die Zeit nicht gesetzt ist. DateStamp meldet immer ein Vielfaches von 50 Takten zurück. Daher ist die so festgestellte Zeit immer auf eine Sekunde genau.

Delay

Aufgabe: Einen Prozeß für die angegebene Zeit anhalten.
Format: Delay(zeit)
D1
Argument: zeit – Integer
Beschreibung: »zeit« gibt die Zeit in Ticks (50stel Sekunden) an, die der Prozeß angehalten werden soll.

DeviceProc

Aufgabe: Meldet die Prozeß-Kennung des Prozesses, der auf das Ein- oder Ausgabe-Device zugreift.
Format: prozeß = DeviceProc(name)
D0 D1
Argument: name – String
Ergebnis: prozeß – Prozeß-Kennung
Beschreibung: DeviceProc meldet die Prozeß-Kennung des Prozesses, der das Device »name« behandelt. Kann DeviceProc keinen solchen Prozeß finden, wird Null ausgegeben. Wenn »name« auf ein aktives gemountetes Device verweist, liefert IoErr() einen Lock auf ein Directory auf diesem Device zurück.

Mit dieser Funktion können Sie feststellen, mit welchem Prozeß Sie kommunizieren müssen, um das entsprechende Device zu steuern.

Exit

Aufgabe: Einen Prozeß beenden.
Format: Exit(returnCode)
D1
Argument: returnCode – Integer
Beschreibung: Exit bewirkt je nach Start des Programmes Verschiedenes. Wurde das Programm aus einem CLI heraus gestartet, beendet Exit das Programm und kehrt in das CLI zurück. Das Argument »returnCode« liefert dann den CLI-Return-Code des Programms an andere CLI-Programme.

Wurde das Programm auf der Workbench gestartet, beendet Exit den gesamten Prozeß. Sein Speicherplatz und Stack werden freigegeben.

6.2.3 Funktionen zum Laden von Programmcode

Execute

Aufgabe: Einen CLI-Befehl ausführen.
Format: erfolg = Execute(anweisungstring, eingabe, ausgabe)
D0 D1 D2 D3
Argument: anweisungstring – String
eingabe – File-Handle
ausgabe – File-Handle
Ergebnis: erfolg – Boolean
Beschreibung: Diese Funktion übernimmt den »anweisungstring«, der einen CLI-Befehl und dessen Argumente enthält, und versucht, die Befehle auszuführen. Der String kann jeden zulässigen Befehl enthalten, der auch direkt in ein CLI eingegeben werden kann. Es kann auch die Ein- und/oder Ausgaberichtung mit »>« und »<« geändert werden.

Der Eingabe-File-Handle ist normalerweise gleich Null, in diesem Fall wird der Befehl im String ausgeführt. Ist das Eingabe-File-Handle nicht gleich Null, werden die Befehle aus dem Eingabe-File ausgeführt, bis dessen Ende erreicht ist. In diesem Fall kann der »anweisungstring« leer sein.

In den meisten Fällen wird das Ausgabe-File-Handle angegeben. Dieses File nimmt die Ausgaben der CLI-Kommandos auf, wenn die Ausgaberichtung nicht bei dem jeweiligen Kommando anders angegeben wird. Wird das Ausgabe-File-Handle nicht angegeben, wird das aktuelle Fenster als Ausgabe verwendet. Denken Sie bitte daran, daß Prozesse, die von Workbench gestartet werden, normalerweise kein aktuelles Fenster haben.

Die Funktion Execute kann dazu verwendet werden, ein neues interaktives CLI zu starten, ähnlich dem Befehl NEWCLI. In diesem Fall sollten Sie Execute mit einem Null-String als Befehlsstring eingeben und ein Input-File-Handle in das neue CLI zeigen lassen. Das

Ausgabe-File-Handle sollte Null sein. Das CLI versucht nun, Befehle aus seinem Fenster zu lesen, die Ausgabe geht ebenfalls in dieses Fenster. Dieses CLI kann nur mit dem Befehl ENDCLI geschlossen werden.

> **Achtung!** Um diesen Befehl ausführen zu können, muß das Programm C:RUN in C: vorhanden sein.

LoadSeg

Aufgabe: Lädt ein ladbares Modul in den Speicher.
Format: segment = LoadSeg(name)
D0 D1
Argument: name – String
Ergebnis: segment – Pointer zu einem Segment.
Beschreibung: Das File »name« muß ein vom Linker erzeugtes ladbares Modul sein. LoadSeg lädt das File, das dabei in seine Code-Segmente zerlegt wird (*scattered loading*). Diese Segmente werden mit dem jeweils ersten Wort miteinander verkettet. Null signalisiert das Ende dieser Kette.

Tritt dabei ein Fehler auf, werden alle schon geladenen Blocks gelöscht und ein Fehler gemeldet.

Wurde das Modul korrekt geladen, liefert LoadSeg einen Pointer auf den Beginn der Liste Segmente. Ist die Arbeit mit dem geladenen Modul beendet, kann es mit UnLoadSeg gelöscht werden. (Zur Benutzung der geladenen Segmente siehe CREATEPROC.)

UnLoadSeg

Aufgabe: Ein vorher mit LoadSeg geladenes Modul löschen.
Format: UnLoadSeg(segment)
D1
Argument: segment – Pointer auf ein Segment
Beschreibung: UnLoadSeg löscht das Modul mit der Segment-Kennung »segment« (die von LoadSeg geliefert wurde). »segment« kann Null sein.

6.3 Kurzübersicht

6.3.1 File-Handling

Close	ein Ein- oder Ausgabe-File schließen.
CreateDir	ein neues Directory erstellen.
CurrentDir	ein Directory zum aktuellen Directory machen.
DeleteFile	ein File oder Directory löschen.
DupLock	einen Lock duplizieren.
Examine	genauere Informationen zu einem Directory oder File erfragen.
ExNext	den nächsten Eintrag eines Directory untersuchen.
Info	Informationen über eine Diskette lesen.
Input	das initiale Eingabe-File erfragen.
IoErr	zusätzliche Informationen zum System feststellen.
InInteractive	feststellen, ob ein File ein virtuelles Terminal ist oder nicht.
Lock	ein File oder Directory für bestimmte Zugriffe sperren.
Open	ein Ein- oder Ausgabe-File öffnen.
Output	das initiale Ausgabe-File erfragen.
ParentDir	das Directory feststellen, in dem ein File oder Directory liegt.
Read	Daten aus einem File lesen.
Rename	ein File oder Directory umbenennen oder in ein anderes Directory bewegen.
Seek	den Schreib-Lese-Zeiger zu einer bestimmten Stelle des Files bewegen.
SetComment	einen Kommentar an ein File anhängen.
SetProtection	die Protect-Flags eines Directorys oder Files setzen.
Unlock	die Sperrung eines Directory oder Files für bestimmte Zugriffe aufheben.
WaitForChar	feststellen, ob innerhalb einer gewissen Zeit ein Zeichen von einem File gelesen werden kann oder nicht.
Write	Daten in ein File schreiben.

6.3.1 Prozeß-Handling

CreateProc	einen neuen Prozeß starten.
DateStamp	Datum und Zeit im internen Format abfragen.
Delay	einen Prozeß eine bestimmte Zeitspanne lang anhalten.
DeviceProc	die Prozeß-Kennung eines Prozesses feststellen, der ein bestimmtes Device behandelt.
Exit	ein Programm beenden.

6.3.3 Funktionen zum Laden von Files

Execute	einen CLI-Befehl ausführen.
LoadSeg	ein ladbares Modul in den Speicher laden.
UnLoadSeg	ein mit LoadSeg geladenens Modul löschen.

Kapitel 7: Der Makro-Assembler

Dieses Kapitel beschreibt den AmigaDOS-Makro-Assembler (von Metacomco). Zusätzlich gibt es einen kurzen Überblick über den Mikroprozessor MC 68000. Dieses Kapitel ist für Programmierer geschrieben, die sich bereits mit Assembler-Programmierung auf einem anderen Computer beschäftigt haben.

7.1 Einführung zum Mikroprozessor MC 68000

Dieser Abschnitt beschreibt den Mikroprozessor 68000. Er soll Ihnen helfen, die später vorgestellten Konzepte zu verstehen. Er setzt voraus, daß Sie bereits Erfahrung im Umgang mit anderen Prozessoren haben.

Der für den Prozessor verfügbare Speicher besteht aus zwei Teilen:

• den internen Registern auf dem Chip selbst und

• dem Hauptspeicher außerhalb des Chips.

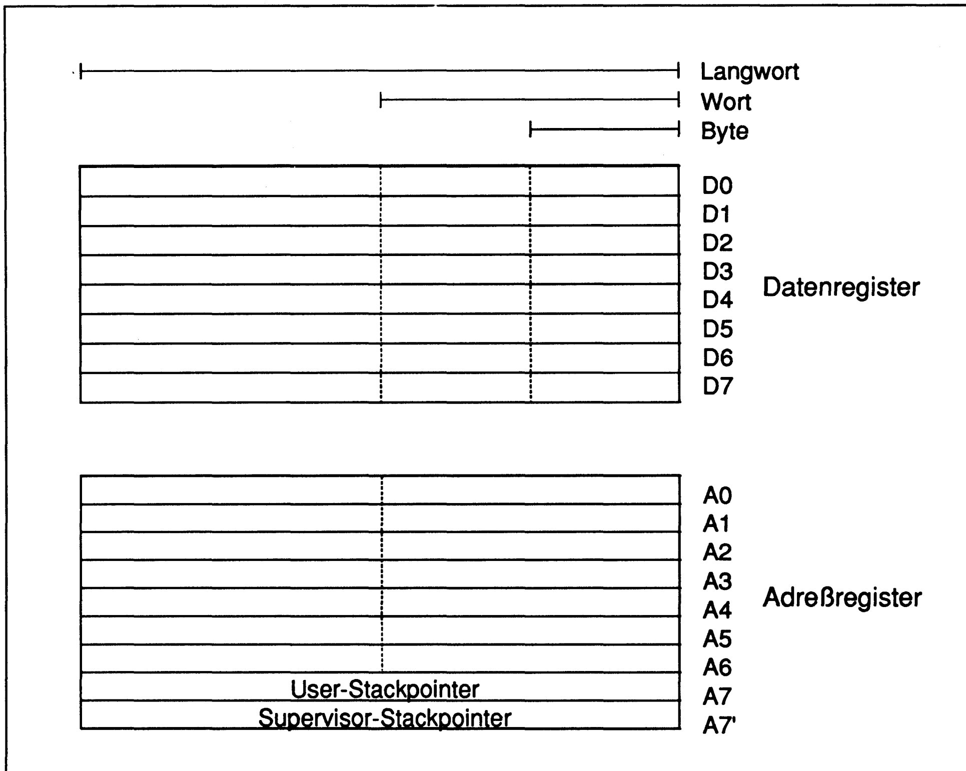


***Bild 7.1:** Die 68000-Prozessor-Register*

Von den 17 Registern des 68000 stehen ständig 16 zur Verfügung. Acht von ihnen sind sogenannte Datenregister, die mit D0 bis D7 bezeichnet werden. Die anderen acht Register werden Adreßregister genannt und mit A0 bis A7 bezeichnet. Jedes Register besteht aus 32 Bit. In vielen Fällen können Sie beliebige Register verwenden, manche Operationen fordern aber explizit einen bestimmten Typ. Zum Beispiel kann jedes Register für interne Operationen mit den Datentypen *Wort* (16 Bit) oder *Langwort* (32 Bit) oder zur indizierten Adressierung des Hauptspeichers verwendet werden. Dagegen wird zur Operation mit *Byte* (8 Bit) ein Datenregister verlangt, zur direkten Adressierung des Hauptspeichers dagegen ein Adreßregister als Stack-Pointer oder Basis-Pointer. Das Register A7 ist der Stack-Pointer. Dabei handelt es sich in Wirklichkeit um zwei Register: im Supervisor-Modus ist es der System-Stack-Pointer, im Anwender-Modus der Benutzer-Stack-Pointer für die Anwendungen.

Der Hauptspeicher besteht aus einer großen Anzahl einzelner Bytes. Jedes Byte hat eine Identifizierungs-Nummer, seine Adresse. Normalerweise (aber nicht immer) beginnen die

Adressen der Bytes mit 0,1,2,3,......n-1. Der Speicher ist dann n Byte groß. Der 68000 kann bis zu 16 Millionen Byte direkt adressieren. Dabei können Operationen mit einem Byte, Wort oder Langwort durchgeführt werden. Ein Wort besteht aus zwei zusammengehörigen Bytes. Dabei hat das erste Byte immer eine gerade Adresse. Ein Langwort besteht aus vier Bytes. Auch dabei hat das erste Byte eine gerade Adresse. Die Adresse des Langwortes ist die Adresse des niedersten geraden Bytes.

Neben den einzelnen Daten, die der Rechner manipulieren soll, enthält der Hauptspeicher auch die Befehle zu den Manipulationen, die er ausführen soll. Jeder Befehl belegt ein bis fünf Worte im Speicher, eines für den eigentlichen Befehl und bis zu vier Operanden. Der Befehl enthält auch die Anzahl an zugehörigen Operanden. Die Operanden enthalten die Adresse der Speicherstellen, deren Werte verarbeitet werden sollen, und die Adressen, wohin das Ergebnis gespeichert werden soll.

Der Prozessor führt einen Befehl im Speicher nach dem anderen aus, so wie Sie ein Musikstück Note für Note vom Blatt abspielen. Ein spezielles Register, der Programm-Counter (PC), enthält die Adresse des als nächsten auszuführenden Befehls. Einige Befehle, Sprünge oder Verzweigungen genannt, werfen diese Ordnung durcheinander. Mit ihnen wird der Rechner angewiesen, als nächstes den Befehl an einer anderen angegebenen Adresse auszuführen. Somit kann der Rechner etwas wiederholt ausführen oder den Fortgang des Programmes von einem Wert abhängig machen.

Um genauere Informationen über den Status des Prozessors zu erhalten, werden Sie das Status-Register (SR) einsetzen.

7.2 Der Aufruf des Assemblers

Das Befehls-Muster für ASSEM lautet:

```
ASSEM "PROG=FROM/A,-O/K,-V/K,-L/K,-H/K,-C/K,-I/K"
```

alternativ kann das Muster so beschrieben werden:

```
ASSEM <ursprungfile>     [-o <objektfile>]
[-I <listingfile>]
[-v <ausgabefile>]
[-h <headerfile>]
[-c <Optionen>]
[-i <include Dir.liste>]
```

Der Assembler erzeugt kein Listingfile oder Objektfile, wenn Sie es nicht besonders anfordern.

Alle Meldungen, die der Assembler während seiner Arbeit erzeugt, werden auf dem Bildschirm ausgegeben, wenn Sie kein Ausgabefile benannt haben.

Um ein File an den Beginn Ihres Ursprungscodes einzubinden, wird die Option -h <filename> verwandt. Den gleichen Effekt hat der Befehl

```
INCLUDE "<filename>"
```

in der ersten Zeile Ihres Quellcodes.

Um die Liste der Directories zu spezifizieren, die vom Assembler auf der Suche nach einem Include-File durchsucht werden sollen, wird die Option »-i« verwendet. Die einzelnen Directories werden in dem Befehl durch ein Leerzeichen, ein Komma oder das (+) getrennt. Verwenden Sie ein Leerzeichen zur Trennung, muß die gesamte Liste der Directories in Anführungszeichen (") stehen. Unix-Anwender müssen zusätzlich allen Anführungszeichen einen Backslash vorstellen (\").

Die Directories werden in der genannten Reihenfolge nach dem File durchsucht. Das File, das vom Assembler eingebunden werden soll, muß im aktuellen oder in einem der mit »-i« bezeichneten Directories stehen. Soll zum Beispiel das File FRED.ASM assembliert und dabei das File INTERN/INCL, das File INCLUDE/ASM und das File EXTERN/INCL eingebunden werden, lauten die drei möglichen Befehle:

```
assem fred.asm -i intern/incl, include/asm, extern/incl

assem fred.asm -i intern/incl+include/asm+extern/incl

assem fred.asm -i "intern/incl include/asm extern/incl"
```

oder, wenn Sie eine SUN unter Unix und das Leerzeichen als Trennung verwenden:

```
assem fred.asm -i \"intern/incl include/asm extern/incl\"
```

Mit der Option »-c« lassen sich noch einige weitere Optionen angeben. Jede besteht aus einem Buchstaben in Groß- oder Kleinschrift, zum Teil gefolgt von einer Zahl. Hier nun die Beschreibung der Optionen:

S erzeugt einen Symbol-Dump als Teil des Objektfiles.

D unterdrückt die Ausgabe lokaler Labels in den Symbol-Dump. (In C beginnt jedes lokale Label mit einem ».«.)

C ignoriert Groß- oder Kleinschreibung bei Labels.

X erzeugt am Ende des Listingfiles eine Cross-Referenz-Tabelle.

Beispiele:

```
ASSEM fred.asm -o fred.o
```

assembliert das File FRED.ASM und erzeugt das Objektfile FRED.O.

```
ASSEM fred.asm -o fred.o -l fred.erst
```

assembliert das File FRED.ASM, erzeugt das Objektfile FRED.O und das Listingfile FRED.ERST.

7.3 Aufbau des Quellcodes

Ein File, das vom Assembler bearbeitet werden soll, besteht aus einer Serie von Zeilen, die folgendes enthalten können:

- Kommentare oder Leerzeilen
- Ausführbare Befehle
- Direktiven an den Assembler

7.3.1 Kommentare

Kommentare können auf drei verschiedenen Wegen in ein Programm eingebracht werden:

1. Geben Sie irgendwo in der Zeile einen Strichpunkt (;) ein. Alles nach diesem Strichpunkt ist Kommentar. Das folgende Beispiel zeigt einen solchen Kommentar:

```
CMPA.L A1, A2 ; Sind die Pointer gleich?
```

2. Geben Sie einen Stern (*) als erstes Zeichen einer Zeile ein. Die gesamte Zeile enthält nur Kommentar. Das folgende Beispiel zeigt einen solchen Kommentar:

```
* Die ganze Zeile enthält Kommentar
```

3. Geben Sie nach einem vollständigen Befehl mindestens ein Leerzeichen ein und dann den Kommentar:

```
MOVEQ #IO,D0 schreibt den Ausgangswert in Register D0
```

Alle Leerzeilen werden vom Assembler als Kommentare aufgefaßt.

7.3.2 Ausführbare Instruktionen

Ausführbare Instruktionen der Quelldatei haben generell dieses Format:

```
[<label>] <opcode> [<operand>[,<operand>]]...[<kommentar>]
```

Um ein Feld von einem anderen zu trennen, können Sie ein Leerzeichen oder die TAB-Taste verwenden. Beide erzeugen ein Trennzeichen. Mehrere Trennzeichen hintereinander sind erlaubt.

7.3.2.1 Das Label-Feld

Ein Label ist ein vom Programmierer definiertes Symbol, das:

a) in der ersten Spalte der Zeile beginnt und vom *opcode* durch mindestens ein Leerzeichen getrennt ist oder

b) irgendwo in einer Zeile steht, dann aber von einem Doppelpunkt (:) beendet wird.

Geben Sie ein Label an, muß es das erste Zeichen einer Zeile sein, das kein Leerzeichen ist. Der Assembler setzt dann den PC (Programmzähler), das ist das Register, das die Adresse des nächsten Befehls im Speicher enthält, auf die Zeile mit dem Label. Labels können allen Befehlen und nahezu allen Direktiven an den Assembler zugewiesen werden, oder sie können auch allein in einer Zeile stehen. In Abschnitt 7.7 finden Sie weitere Informationen zu Labels bei Direktiven.

Jedes Label darf nur einmal vergeben werden. Der Name eines Labels darf keine Makro-Namen, keine Device-Namen, keine Befehle oder Register-Bezeichnungen enthalten.

7.3.2.2 Lokale Labels

Die lokalen Variablen wurden zusätzlich zu den normalen Labels von Motorola eingeführt. Sie werden in der Form »nnn$« angegeben und sind nur zwischen zwei normalen Labels gültig. Hier ein Beispiel:

```
Label   Opcode   Operanden
FOO:    MOVE.L   D6,D0
1$:     MOVE.B   (A0)+,(A1)+
        DBRA     D0,1$
        MOVEQ    #20,D0
BAA:    TRAP     #4
```

In diesem Fall ist das Label »1$« nur erreichbar, wenn es nach der Zeile mit dem Label FOO und vor der Zeile mit dem Label BAA verwendet wird. Der Name »1$« kann in diesem Fall zwischen zwei anderen Labels jederzeit wieder verwendet werden.

7.3.2.3 Das Opcode-Feld

Das Feld Opcode kommt nach dem Label und ist von diesem durch mindestens ein Leerzeichen getrennt. In diesem Feld können stehen:

a) Opcodes für den MC 68000, wie im *MC6800 User Manual* definiert
b) Direktiven an den Assembler
c) Makro-Aufrufe

Zusätzlich können den Opcodes, die mehr als eine Datengröße verarbeiten können, noch Kennungen für die jeweilige Datengröße übergeben werden. Diese werden vom Operator nur durch einen Punkt (.) getrennt.

Diese Kennungen sind:

B Größe ist ein Byte (8 Bit)
W Größe ist Wort (16 Bit)
L Größe ist Langwort (32 Bit) oder Kennung für langen Sprung
S Kennung für kurzen Sprung

Die angegebene Größe muß natürlich zum jeweiligen Befehl passen.

7.3.2.4 Das Operanden-Feld

Falls vorhanden, enthält dieses Feld einen oder mehrere Operanden zu dem Befehl oder der Direktive. Der Operand wird durch mindestens ein Leerzeichen vom Opcode getrennt. Sind mehrere Operanden angegeben, werden diese durch Kommata getrennt. Das Operanden-Feld endet mit einem Leerzeichen oder mit dem Zeichen für *neue Zeile*, das erscheint, wenn <Return> betätigt wird. Leerzeichen zwischen den Operanden müssen in diesem Fall nicht eingegeben werden.

7.3.2.5 Das Kommentar-Feld

Dieses Feld wurde bereits in Abschnitt 7.3.1 besprochen.

7.4 Ausdrücke

Ein Ausdruck ist eine Kombination von Symbolen, Konstanten, algebraischen Operatoren und Klammern, mit denen Sie Befehle oder Direktiven für den Assembler erstellen.

7.4.1 Operatoren

An Operatoren stehen zur Verfügung (aufgelistet nach der Wertigkeit):

1. Minus als Vorzeichen und logisches NOT (– und ~)
2. LinksSHIFT und RechtsSHIFT (<< und >>)
3. Logisches UND und logisches ODER (& und !)
4. Multiplikation und Division (* und /)
5. Addition und Subtraktion (+ und –)

Um die Wertigkeit eines Operators zu umgehen, wird der ganze Term in Klammern eingeschlossen. Operationen gleicher Wertigkeit werden von links nach rechts ausgeführt. Bitte beachten Sie, daß ein Leerzeichen als Trennung zwischen den Feldern steht.

7.4.2 Operand/Operator-Kombinationen

In der folgenden Tabelle wird die Zulässigkeit eines Operators zu den Operanden dargestellt. Die Tabelle zeigt alle möglichen Operand/Operator-Kombinationen auf. Dabei steht »A« für

einen absoluten Operanden und »R« für einen relativen. Ein »x« zeigt einen Fehler an und »op« steht für Operator. Das Vorzeichen Minus und die logischen Operationen sind nur für absolute Operanden zulässig.

<table>
<tr><th rowspan="2">Operatoren</th><th colspan="4">Operanden</th></tr>
<tr><th>A op A</th><th>R op R</th><th>A op R</th><th>R op A</th></tr>
<tr><td>+</td><td>A</td><td>x</td><td>R</td><td>R</td></tr>
<tr><td>-</td><td>A</td><td>A</td><td>A</td><td>R</td></tr>
<tr><td>*</td><td>A</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>/</td><td>A</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>&</td><td>A</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>|</td><td>A</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>>></td><td>A</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td><<</td><td>A</td><td>x</td><td>x</td><td>x</td></tr>
</table>

Tabelle 7.1: *Zulässige Operand/Operator-Kombinationen*

7.4.3 Symbole

Ein Symbol ist ein String aus bis zu 30 Zeichen. Das erste Zeichen eines Symboles muß eines der folgenden sein:

- Ein alphanumerisches Zeichen (A–Z oder a–z)
- Ein Unterstreichungsstrich (_)
- Ein Punkt (.)

Die restlichen Zeichen können außerdem noch aus Ziffern (0–9) bestehen. Groß- oder Kleinschrift wird beachtet! Das Symbol »fred« ist nicht gleich dem Symbol »FRED« oder »FRed«! Dies gilt jedoch nicht, wenn Sie beim Aufruf des Assemblers die Option -C angegeben haben. Alle Opcodes, Befehle, Argumente oder Direktiven können in Groß- oder Kleinbuchstaben eingegeben werden. Ein Symbol kann aus 30 Zeichen bestehen, alle werden beachtet. Geben Sie mehr als 30 Zeichen ein, werden die überzähligen abgeschnitten und eine Fehlermeldung ausgegeben. Ein einem Register-Namen mit EQUR gleichgestelltes Label wird ebenso in Groß- oder Kleinschrift erkannt. Die Namen von Befehlen, Direktiven, Devices, Registern und die Sonderzeichen CCR, SR, SP und USP dürfen Sie nicht als Symbole verwenden. Drei Typen von Symbolen sind möglich:

Absolut
a) Das Symbol wurde mit SET oder EQU einem absoluten Wert gleichgestellt.

Relativ
a) Das Symbol wurde mit SET oder EQU einem relativen Wert gleichgestellt.
b) Das Symbol wurde als Label benutzt.

Register
a) Das Symbol wurde durch EQUR als Register-Name gesetzt.

Das besondere Symbol »*« hat den Wert und Typ des aktuellen Programm-Zählers, also der Adresse des gerade ausgeführten Befehls oder der Direktive.

7.4.4 Zahlen

Ziffern werden als Teil eines Ausdrucks oder als einzelner Wert angegeben. Zahlen haben immer absolute Werte und entsprechen einem der folgenden Formate:

Dezimal:
(eine Reihe dezimaler Ziffern)
Beispiel: 12345

Hexadezimal:
($ gefolgt von hexadezimalen Zeichen)
Beispiel: $89AC2

Oktal:
(@ gefolgt von oktalen Ziffern)
Beispiel: @73413

Binär:
(% gefolgt von Einsen und Nullen)
Beispiel: %01101101

ASCII-Literale:
(Bis zu vier ASCII-Zeichen in »´« eingeschlossen)
Beispiel: ´ABCD´ ´*´

ASCII-Strings mit weniger als vier Zeichen werden rechtsbündig verwendet und mit Null auf vier Zeichen ergänzt.

Der Apostroph innerhalb eines Strings wird durch einen zweiten kenntlich gemacht. Ein Beispiel ist:

´Holger´ ´s´

7.5 Adressierungs-Arten

Die genaue Beschreibung der unterschiedlichen Adressierungs-Arten des MC68000 finden Sie in jedem guten Handbuch zu diesem Prozessor. Die folgende Tabelle gibt einen Überblick:

Adressierung	Beschreibung und Beispiele
Dn	Daten-Register direkt Beispiel: MOVE D0,D1
An	Adreß-Register direkt Beispiel: MOVEA A0,A1
(An)	Adreß-Register indirekt Beispiel: MOVE D0,(A1)
(An)+	Adreß-Register indirekt mit Postinkrement Beispiel: MOVE (A7)+,D0
-(An)	Adreß-Register indirekt mit Predekrement Beispiel: MOVE D0,-(A7)
a(An)	Adreß-Register indirekt, verschoben Beispiel: MOVE 20(A0),D1
a(An,Xn)	Adreß-Register indirekt, indiziert Beispiele: MOVE 0(A0,D0),D1 MOVE 120(A0,D6.W),D4
a	absolut, kurz (16 Bit) Beispiel: MOVE $1000,D0
a	absolut, lang (32 Bit) Beispiel: MOVE $10000,D0
r	Programm-Zähler (PC) relativ mit Verschiebung Beispiel: MOVE ABC,D0 (ABC ist relativ)
r(Xn)	PC relativ, indiziert Beispiel: MOVE ABC(D0.L),D1 (ABC ist relativ)
#a	Daten literal Beispiel: MOVE #1234,D0
USP CCR SR	Spezieller Adreß-Modus Spezieller Adreß-Modus Spezieller Adreß-Modus Beispiele: MOVE A0,UPS MOVE D0,CCR MOVE D1,SR

Tabelle 7.2: Adressierungs-Arten

Adressierungen beziehen sich immer auf ein bestimmtes Byte. Befehle und Operationen mit der Größe Wort und Langwort benötigen jedoch mehr als ein Byte. Diese Daten müssen bei einer gerade Adresse beginnen. In der obigen Tabelle steht »Dn« für ein Daten-Register D0-D7, ein »An« für ein Adreß-Register A0-A7, SP und PC, ein »a« für einen absoluten Wert, ein »r« für einen relativen Wert und »Xn« steht für An oder Dn mit den Optionen ».W« oder ».L«.

7.6 Verschiedene Befehlstypen

Einige Befehle (zum Beispiel ADD, CMP) besitzen spezielle Varianten. Adreßvarianten beziehen sich auf Adreßregister als Ziel. Immediate- und Quick-Varianten arbeiten mit literalen Daten. Memory-Varianten benötigen Postinkrement-Adressen für beide Operanden.

Um eine dieser Varianten einzusetzen, werden dem Mnemonic die Zusätze »A«dreß, »Q«uick, »I«mmediate oder »M«emory angehängt. Dann benutzt der Assembler die jeweilige Sonderform des Befehls, wenn vorhanden. Ansonsten gibt er eine Fehlermeldung aus.

Haben Sie keine der Varianten definiert, verwendet der Assembler automatisch diese Formen, wenn das möglich ist. Das gilt nicht für die Variante »Q«. Der Assembler wandelt zum Beispiel die Form

```
ADD.L A2,A1
```

automatisch in die folgende kompaktere Form um:

```
ADDA.L A2,A1
```

7.7 Direktiven

Alle Assembler-Direktiven (mit Ausnahme von DC und DCB) sind Befehle an den Assembler, sie werden nicht direkt in den Objektcode umgesetzt. Zu Beginn dieses Abschnittes finden Sie eine Tabelle mit allen Direktiven (Tabelle 7.3), geordnet nach Funktionen; später werden alle Direktiven genauer beschrieben. Diese Beschreibungen sind ebenfalls nach Funktionen geordnet.

Beachten Sie bitte, daß nicht allen Direktiven ein Label zugeordnet werden darf. Zum Beispiel ist zu EQU ein Label erlaubt, möglich ist es bei RORG, nicht zulässig aber bei LLEN oder TTL. Die Zulässigkeit eines Labels wird in der genauen Beschreibung vermerkt.

Direktive	Beschreibung
Assembler-Kontrolle	
SECTION	Programm-Abschnitt
RORG	Verschiebbarer Anfang
OFFSET	Definiert Offsets
END	Ende des Programmes
Symbol-Definition	
EQU	Weist konstanten Wert zu
EQUR	Weist Register konstanten Wert zu
REG	Weist konstanten Wert zu
SET	Weist variablen Wert zu
Daten-Definition	
DC	Definiert Konstante
DCB	Definiert konstanten Block
DS	Definiert Speicherplatz-Reservierung
Listing-Kontrolle	
PAGE	Seitenvorschübe im Listing: ein
LIST	Listing einschalten
NOLIST(NOL)	Listing ausschalten
SPC n	fügt n Leerzeilen ein
NOPAGE	Seitenvorschub im Listing: aus
LLEN n	Setzt die Anzahl der Zeichen pro Zeile (60<=n<=132)
PLEN n	Setzt die Anzahl der Zeilen pro Seite (24<=n<=100)
TTL	Setzt Programm-Name (max. 40 Zeichen)
NOOBJ	Verhindert Ausgabe von Objektcode
FAIL	Erzeugt Assembler-Fehler
FORMAT	Keine Funktion
NOFORMAT	Keine Funktion

***Tabelle 7.3:** Assembler-Direktiven (Fortsetzung nächste Seite)*

Direktive	Beschreibung
Bedingte Assemblierung	
CNOP	Bedingtes NOP zur Angleichung
IFEQ	Assembliert, wenn Ausdruck 0
IFNE	Assembliert, wenn Ausdruck nicht 0
IFGT	Assembliert, wenn Ausdruck >0
IFGE	Assembliert, wenn Ausdruck >=0
IFLT	Assembliert, wenn Ausdruck <0
IFLE	Assembliert, wenn Ausdruck <=0
IFC	Assembliert, wenn Strings gleich
IFNC	Assembliert, wenn Strings ungleich
IFD	Assembliert, wenn Symbol definiert
IFND	Assembliert, wenn Symbol nicht definiert
ENDC	Ende der bedingten Assemblierung
Direktiven zu Makros	
MAKRO	Definiert Makro-Name
NARG	Spezielles Symbol
ENDM	Ende des Makro
MEXIT	Verläßt Makro vorzeitig
Externe Symbole	
XDEF	Definiert externes Symbol
XREF	Nimmt Bezug auf externes Symbol
Allgemeine Direktiven	
INCLUDE	Bindet File in Quellcode
MASK2	Keine Funktion
IDNT B	Benennt Programm-Einheit

Tabelle 7.3: *Assembler-Direktiven (Fortsetzung)*

7.7.1 Direktiven zur Assembler-Kontrolle

SECTION

Muster: [<label>] SECTION <name>[,<typ>]

Beschreibung: Diese Direktive weist den Assembler an, den Zähler zum letzten Befehl der benannten Sektion zurückzusetzen. (Der Zähler wird Null, wenn er zum ersten Mal benutzt wird.)

<name> ist ein String, eventuell in »´« eingeschlossen.
<typ> ist, wenn angegeben, eines der folgenden Schlüsselworte:

CODE (der Default) zeigt an, daß die Sektion verschiebbaren Code enthält.
DATA zeigt an, daß die Sektion nur fest definierte Daten enthält.
BSS zeigt an, daß die Sektion variable Daten enthält.

Der Assembler kann bis zu 255 Sektionen verwalten. Der Assembler beginnt mit einer nicht benannten Sektion vom Typ CODE. Der Assembler weist das optionale Argument <label> nach der Ausführung von SECTION dem PC zu. Ist eine Sektion unbenannt, wird das Schlüsselwort CODE zum Namen dieser Sektion.

RORG

Muster: [<label>] RORG <absolutwert>

Beschreibung: Die Direktive RORG ändert den Wert des PC zu <absolutwert> Bytes hinter den Start der aktuellen verschiebbaren Sektion. Der Assembler weist den folgenden Befehlen verschiebbare Adressen zu. Um Adressierungen in verschiebbaren Sektionen vornehmen zu können, muß die Adressierung *PC relativ mit Verschiebung* angewandt werden. Die Wertzuweisung an <label> ist ansonsten gleich der bei SECTION.

OFFSET

Muster: OFFSET <absolutwert>

Beschreibung: Um eine Tabelle von Offset-Werten mit der Direktive DS anzulegen, die bei <absolutwert> beginnt, benötigen Sie die OFFSET-Direktive. In einer OFFSET-Tabelle definierte Symbole werden intern verwaltet, dürfen jedoch keine Befehle enthalten, die Opcodes erzeugen. Um eine OFFSET-Sektion zu beenden, können RORG, OFFSET, SECTION oder END verwendet werden.

END

Muster: [<label>] END

Beschreibung: Die END-Direktive teilt dem Assembler das Ende des Quellcodes mit. Alle weiteren Befehle oder Direktiven werden ignoriert. Erreicht der Assembler im ersten Durchlauf ein END, beginnt er den zweiten. Wird vor dem END die End-of-File-Marke erreicht, wird eine Fehlermeldung ausgegeben.

Geben Sie ein Label ein, wird dem PC der Wert des Labels zugewiesen, bevor END ausgeführt wird.

7.7.2 Direktiven zur Symbol-Definition

EQU

Muster: <label> EQU <ausdruck>

Beschreibung: Der Wert des Operanden <ausdruck> wird dem Label <label> zugewiesen. Die Zuweisung ist absolut, Sie dürfen dieses Label innerhalb des Programmes kein zweites Mal definieren.

Sie sollten in <ausdruck> nicht auf andere, noch folgende Definitionen verweisen. Auf die mit EQU zugewiesenen Symbole darf vor der Definition nicht zugegriffen werden.

EQUR

Muster: <label> EQUR <register>

Beschreibung: Mit dieser Direktive wird ein Symbol <label> einem Register des Prozessors gleichgestellt. Diese Direktive ist nur für Adreß- und Daten-Register zulässig. Die Definition ist absolut, das Label darf kein zweites Mal definiert werden. Der Assembler unterscheidet bei dem Label nicht zwischen Groß- und Kleinschreibung.

Auf die mit EQUR zugewiesenen Symbole darf vor der Definition nicht zugegriffen werden.

REG

Muster: <label>REG<registerliste>

Beschreibung: Der Assembler weist Label eine Liste von Registern nach dem mit MOVEM verwandten Muster zu. Dabei ist <registerliste> von folgender Form:

```
R1[-R2][/R3[-R4]]...
```

SET

Muster: <label> SET <ausdruck>

Beschreibung: Mit SET wird dem <label> der Wert von <ausdruck> zugewiesen. SET unterscheidet sich von EQU dadurch, daß die Zuweisung des Wertes zum Symbol nicht dauerhaft ist ist. Sie kann später im Programm jederzeit verändert werden.

Sie sollten in <ausdruck> nicht auf andere noch folgende Definitionen verweisen. Auf die mit SET zugewiesenen Symbole darf vor der Definition nicht zugegriffen werden.

7.7.3 Direktiven zur Daten-Definition

DC

Muster: [<label>] DC[.<größe>] <liste>

Beschreibung: Mit DC wird eine Liste von Konstanten in den Speicher plaziert. Mehrere Konstanten können, durch Kommata getrennt, angegeben werden. Die Werte müssen dabei jedoch in der mit <größe> angegebenen Anzahl an Bits speicherbar sein. Wird <größe> nicht angegeben, ist ».W« voreingestellt.

Geben Sie als <größe> ».B« an, steht zur Speicherung auch die Option ASCII-String zur Verfügung. Das ist eine beliebig lange Folge von ASCII-Zeichen, eingefaßt in Apostrophe »´«. Soll der Apostroph innerhalb des Strings verwendet werden, müssen zwei hintereinander eingegeben werden. Ist als <größe> ».W« oder ».L« angegeben, füllt der Assembler den Wert bis zur vollen Größe auf.

DCB

Muster: [<label>] DC[.<größe>] <absolutwert>,<ausdruck>

Beschreibung: Mit DCB wird eine Anzahl <absolutwert> von Bytes, Worten oder Langworten (abhängig von <größe>) in den Speicher plaziert. Jedes dieser Bytes, Worte oder Langworte erhält denselben Inhalt, nämlich den Wert von <ausdruck>. DCB<.größe> n, <ausdruck> entspricht n Wiederholungen der Direktive DC<größe> <ausdruck>.

DS

Muster: [<label>] DS[.<größe>] <absolutwert>

Beschreibung: DS reserviert <absolutwert> Bytes, Worte oder Langworte (abhängig von <größe>). Wird <größe> nicht angegeben, ist ».W« eingestellt. DS initialisiert diesen Speicher jedoch nicht. Die folgende Direktive ergibt zum Beispiel einen reservierten Speicherbereich von 1024 Byte.

```
DS.L 256
```

7.7.4 Listing-Kontrolle

PAGE

Muster: PAGE

Beschreibung: PAGE beginnt eine neue Seite im Assembler-Listing. PAGE selbst erscheint nicht im Ausgabe-Listing.

LIST

Muster: LIST

Beschreibung: Die Direktive LIST weist den Assembler an, ein formatiertes Listing der folgenden Instruktionen in das Listing-File zu schreiben. Der Befehl wird ausgeführt, bis der Assembler auf ein NOLIST oder END trifft. Diese Direktive ist nur wirksam, wenn beim Aufruf des Assemblers ein Listing-File angegeben wurde. die Direktive LIST selbst erscheint nicht im Ausgabe-Listing.

NOLIST oder NOL

Muster: NOLIST

Beschreibung: NOL oder NOLIST hebt die Wirkung von LIST auf – die folgenden Instruktionen erscheinen nicht im Ausgabe-Listing. Auch diese Direktive selbst erscheint nicht im Ausgabe-Listing.

SPC

Muster: SPC <zahl>

Beschreibung: SPC erzeugt <zahl> Leerzeilen im Assembler-Listing. Die Direktive SPC selbst erscheint nicht im Ausgabe-Listing.

NOPAGE

Muster: NOPAGE

Beschreibung: Mit NOPAGE wird die Ausgabe von Seitenvorschüben und von Kopfzeilen im Assembler-Listing unterdrückt.

LLEN

Muster: LLEN <zahl>

Beschreibung: Mit LLEN wird die Länge einer Zeile im Assembler-Listing-File eingestellt. Der Wert <zahl> muß zwischen 60 und 132 liegen, er kann nur einmal eingestellt werden. Grundeinstellung ist 132 Zeichen/Zeile. Die Direktive LLEN erscheint nicht im Ausgabe-Listing.

PLEN

Muster: PLEN <zahl>

Beschreibung: Mit PLEN wird die Anzahl der Zeilen pro Seite des Assembler-Listings eingestellt. <zahl> muß zwischen 24 und 100 liegen. Der Wert kann nur einmal verändert werden. Grundeinstellung ist 60 Zeilen pro Seite.

TTL

Muster: TTL <namestring>

Beschreibung: TTL gibt dem assemblierten Programm einen Namen, der in der Kopfzeile des Assembler-Listings ausgedruckt wird. Der Name beginnt mit dem ersten Zeichen nach TTL, das kein Leerzeichen ist, und darf nicht länger als 40 Zeichen sein. Die Direktive TTL erscheint nicht im Ausgabe-Listing.

NOOBJ

Muster: NOOBJ

Beschreibung: Mit NOOBJ wird die Ausgabe des Objektcodes in ein File verhindert, auch wenn Sie beim Aufruf des Assemblers ein Objektcode-File angegeben haben.

FAIL

Muster: FAIL

Beschreibung: Fail weist den Assembler an, einen Fehler in dieser Eingabezeile anzuzeigen.

FORMAT

Muster: FORMAT

Beschreibung: Der Assembler akzeptiert diese Direktive, führt jedoch keine Aktion aus. FORMAT wurde implementiert, um die Kompatibilität mit anderen Assemblern zu gewährleisten.

NOFORMAT

Muster: NOFORMAT

Beschreibung: Der Assembler akzeptiert diese Direktive, führt jedoch keine Aktion aus. NOFORMAT wurde implementiert, um die Kompatibilität mit anderen Assemblern zu gewährleisten.

7.7.5 Bedingte Assemblierung

CNOP

Muster: [<label>] CNOP <zahl>,<zahl>

Beschreibung: Diese Direktive gehört nicht zum Standard-Befehlssatz eines MC68000-Assemblers. Sie erlaubt es, einen Bereich des erzeugten Codes an einer Grenze auszurichten. Jede Datenstruktur und jeder Einsprungpunkt kann so an einer Langwortgrenze ausgerichtet werden.

Der erste Ausdruck repräsentiert einen Versatz, der zweite die gewünschte Ausrichtung. Der Code wird mit dem angegebenen Versatz bei der nächsterreichbaren Grenze ausgerichtet. Die folgende Anweisung richtet also den Code am nächsten Langwort aus (setzt den PC also an die nächste durch vier teilbare Adresse):

```
CNOP 0,4
```

Die nächste Anweisung hingegen richtet den erzeugten Code an der Adresse 2 Byte hinter der nächsten Langwortgrenze aus:

```
CNOP 2,4
```

IFxx

Muster: IFxx '<absolutwert>

Beschreibung: Mit IFxx wird die bedingte Assemblierung gesteuert. Alle nach dieser Direktive folgenden Befehle werden nur assembliert, wenn die Bedingung erfüllt ist. Der Bereich der bedingten Assemblierung geht bis zur Direktive ENDC. Dahinter fährt die Assemblierung auf alle Fälle wieder fort. Bedingungen können auch geschachtelt werden. Dabei muß aber beachtet werden, daß die jeweils zuletzt aufgestellte Bedingung zuerst mit einem ENDC beendet werden muß. Jede Bedingung muß mit einem eigenen ENDC versehen sein.

Für »xx« (die eigentliche Bedingung) können Sie verwenden:

```
EQ Assembliert, wenn Ausdruck = 0
NE Assembliert, wenn Ausdruck <> 0
GT Assembliert, wenn Ausdruck >0
GE Assembliert, wenn Ausdruck >=0
LT Assembliert, wenn Ausdruck <0
LE Assembliert, wenn Ausdruck <=0
```

IFC, IFNC

Muster: IFC <string>,<string>
IFNC <string>,<string>

Beschreibung: IFC assembliert die folgenden Anweisungen (bis zum ENDC) nur, wenn <string1> und <string2> gleich sind, IFNC assembliert, wenn sie nicht gleich sind. Dabei ist jeder <string> eine Kette von ASCII-Zeichen, der von Apostrophen begrenzt wird. Ist die Bedingung NICHT erfüllt, wird der Teil des Quellcodes übersprungen, der zwischen dem IF(N)C und der Direktive ENDC steht.

IFD und IFND

Muster: IFD <symbolname>
IFND <symbolname>

Beschreibung: Der Bereich bis zum nächsten ENDC wird assembliert, wenn das Symbol <symbolname> definiert (IFD) oder nicht definiert (IFND) ist.

ENDC

Muster: ENDC

Beschreibung: Um den Bereich der bedingten Assemblierung abzuschließen, wird ENDC eingesetzt. Damit werden alle acht oben beschriebenen Direktiven zur bedingten Assemblierung (die mit IF... beginnen) beendet.

7.7.6 Direktiven zu Makros

MAKRO

Muster: <label> MAKRO

Beschreibung: Mit MAKRO beginnt die Definition eines Makros. ENDM beendet eine Makro-Definition. Ein Label muß angegeben werden, das der Assembler als Makro-Namen verwendet. Beim Aufruf des Labels als Operand wird statt des Labels der Quellcode zwischen dem entsprechenden MAKRO und ENDM eingesetzt. Ein Makro kann aus Opcodes, Assembler-Direktiven oder weiteren Makros bestehen. Ein (+) im Listing markiert den Code, der von einem Makro erstellt wurde. Jedem Makro-Aufruf können Argumente übergeben werden. Sie werden, durch Kommata getrennt, hinter dem Makro-Namen eingegeben. Enthält ein Argument ein Leerzeichen (zum Beispiel ein String), muß das ganze Argument zwischen die Zeichen *Kleiner als* (<) und *Größer als* (>) gesetzt werden.

Der Assembler speichert den Quellcode, den Sie in das Makro schreiben (nach der Direktive MAKRO und vor der Direktive ENDM), als Makro. Das Makro kann jeden normalen Quellcode enthalten. Zusätzlich hat das Symbol »\« (Backslash) Bedeutung. Ein <Backslash> gefolgt von einer Zahl n versteht der Assembler als Aufforderung, das n-te Argument an dieser Stelle in den Code einzubauen. Wurde das n-te Argument beim Aufruf des Makros nicht angegeben, wird nichts eingesetzt. Ein <Backslash> gefolgt vom Zeichen <@> weist den Assembler an, den Text ».nnn« zu erstellen, wobei nnn die Anzahl der

Aufrufe einer »\@«-Kombination ist. Dies wird normalerweise eingesetzt, um gleiche Labels innerhalb eines Makros zu erzeugen.

Makro-Definitionen dürfen nicht verschachtelt werden. Jedoch kann in einem Makro jederzeit ein vorher definiertes Makro aufgerufen werden. Es dürfen allerdings *nur* zehn Makro-Ebenen auf diese Weise ineinander eingebaut werden.

Der Einbau eines Makros wird beendet, wenn das Ende des Makros erreicht, oder die Direktive MEXIT gefunden wird.

Ein Makro unterscheidet sich von einem Unterprogramm einer höheren Programmiersprache deutlicher, als es im ersten Moment den Eindruck erweckt. Ein Unterprogramm wird angesprungen, ausgeführt, und dann wird die Programmausführung an dem Punkt des Hauptprogrammes weitergeführt, der nach dem Unterprogramm-Aufruf kommt. Ein Makro dagegen ist nur eine gute Kopierfunktion. Der gesamte Quellcode des Makros wird bei jedem Aufruf gleichsam an anderer Stelle in den Quellcode hineinkopiert. (Zeitraubende) Sprünge innerhalb des Programmes sind dazu nicht nötig.

NARG

Muster: NARG
Beschreibung: NARG wird nur in Makros verwendet. NARG entspricht immer der Nummer des letzten Argumentes, das beim Aufruf eines Makros angegeben wurde. Außerhalb eines Makros hat NARG den Wert Null.

ENDM

Muster: ENDM
Beschreibung: Beendet die Definition eines Makros.

MEXIT

Muster: MEXIT
Beschreibung: Mit der Kombination MEXIT (und bedingter Assemblierung) kann ein Makro vorzeitig – das heißt vor Erreichen der Direktive ENDM – abgebrochen werden.

7.7.7 Externe Symbole

XDEF

Muster: XDEF <label>[,<label>...]
Beschreibung: XDEF folgen ein oder mehrere absolute oder verschiebbare Labels. Jedes hier genannte Label erzeugt eine externe Symbol-Definition. Auf diese Symbole kann sich ein anderes Modul (das zum Beispiel in einer höheren Programmiersprache geschrieben wurde) beziehen. Die entsprechende Verbindung stellt der Linker her. Verwenden Sie diese Direktive oder auch XREF, kann der vom Assembler erzeugte Objektcode nicht ohne weiteres Linken ausgeführt werden.

XREF

Muster: XREF <label>[,<label>...]

Beschreibung: Ein oder mehrere Labels müssen hinter XREF eingegeben werden. Diese Labels dürfen an anderer Stelle im Programm nicht definiert werden. Diese Labels bezeichnen Adressen in anderen Modulen, die erst später vom Linker berechnet und festgelegt werden.

Externe Symbole werden normalerweise folgendermaßen eingesetzt: Um eine Routine innerhalb eines Programm-Segmentes als externe Routine zu definieren, wird ein Label an ihren Anfang gesetzt. Dieses Label wird dann der Direktive XDEF *nach außen gereicht.* Ein anderes Programm ruft diese Routine auf, indem es mit XREF ein Label definiert und das so deklarierte Label anspringen kann, als stände es im selben Programm.

7.7.8 Allgemeine Direktiven

INCLUDE

Muster: INCLUDE "<filename>"

Beschreibung: Die Direktive INCLUDE erlaubt das Einbinden externer Files in den Quellcode. Der Filename wird als Operand übergeben. Der Assembler verwendet den Inhalt dieses Files dann, als würde er statt der INCLUDE-Direktive in dem gerade assemblierten File stehen. INCLUDE kann bis zu drei Ebenen tief verschachtelt werden, wenn die Filenamen, wie gezeigt, in Anführungszeichen gesetzt werden.

INCLUDE wird besonders dann benutzt, wenn Makros oder EQU-Zuweisungen in gleicher Form in mehreren Programmen eingebaut werden sollen. Diese Definitionen werden dazu zweckmäßigerweise in ein File geschrieben, das dann in alle Programme mit INCLUDE eingebunden wird. Zu Beginn dieses Files können Sie die Direktive NOLIST, zum Ende LIST einsetzen, um die Ausgabe dieser sich stets wiederholenden Zeilen in das Listfile zu verhindern.

Das einzubindende File wird zuerst im aktuellen Directory, dann in der beim Aufruf des Assemblers übergebenen »-I Dir.liste« gesucht.

MASK2

Muster: MASK2

Beschreibung: Der Assembler akzeptiert diese Direktive, führt jedoch keine Aktion aus.

IDNT

Muster: IDNT <string>

Beschreibung: Eine Programm-Einheit, die aus einer oder mehreren Sektionen besteht, muß einen Namen haben. Dieser wird mit der Direktive IDNT übergeben. Findet der Assembler diese Direktive nicht, bekommt das Programm einen Null-String als Namen.

Kapitel 8: Der Linker

Dieses Kapitel beschreibt ALINK, den Linker des AmigaDOS. Der Linker bindet ein oder mehrere Eingabe-Files zu einem (als Programm ausführbaren) Binär-File. Der Linker ist auch fähig, Programme, die mit dem Overlay-Prinzip arbeiten, zu erzeugen.

8.1 Einführung

ALINK produziert aus einem oder mehreren Eingabe-Files ein binäres Ausgabe-File. Die Eingabe-Files, auch Objekt-Files genannt, können Informationen über externe Symbole erhalten. Objekt-Files werden mit einem Assembler oder Compiler erstellt. Bevor der Linker das Ausgabe- oder auch ausführbare File erstellt, werden alle Bezüge auf Programmteile in Symbole umgewandelt.

Der Linker erzeugt außerdem, wenn das gewünscht wird, eine *Cross-Referenz-Tabelle* und ein Hilfs-File, *Link-Map* genannt.

In den Linker ist ein sogenannter *Overlay-Supervisor* integriert. Mit diesem Programm können Module, die in einer Vielzahl von Programmiersprachen geschrieben wurden, zu einem Overlay-Programm zusammengebunden werden.

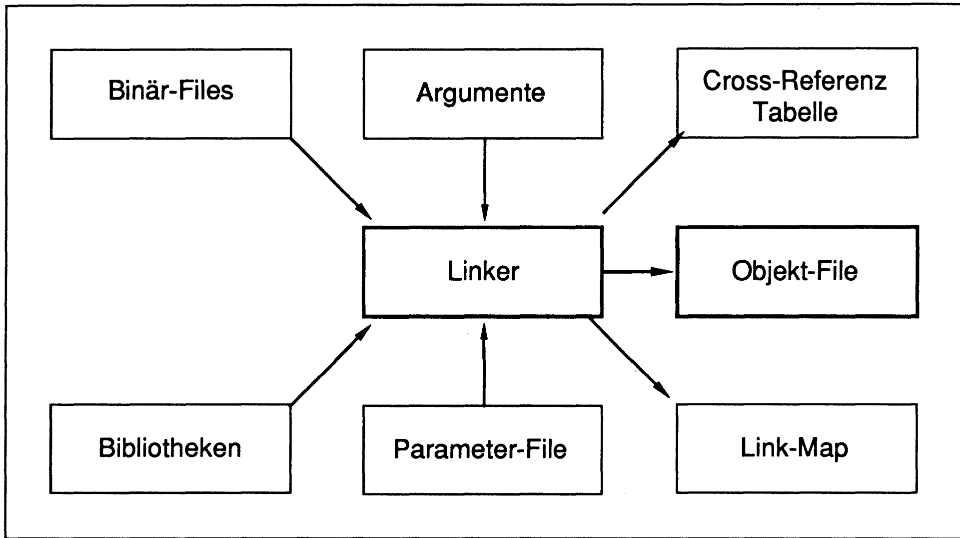


Bild 8.1: Das Prinzip eines Linkers

Der Linker kann seine Argumente auf zwei verschiedenen Wegen erhalten:

1. Aus der *Befehlszeile*. Die Informationen, die der Linker zur Arbeit braucht, werden als Argumente übergeben, wie bei CLI-Kommandos üblich.

2. Aus einem *Parameter-File*. Diese Alternative bietet sich an, wenn ein Programm mehrmals gelinkt werden soll. Alle Parameter werden dann in dem Parameter-File gespeichert. Beim Aufruf des Linkers werden alle Parameter aus dem File übernommen.

Die beiden Methoden können auf drei Arten von Input-Files angewandt werden:

1. *Primär binärer Input*: Ein oder mehrere binäre Files bilden den Input zum Linker. Diese Files werden immer in das ladbare File übernommen, dieses Input-File muß immer vorhanden sein.

2. *Overlay-Files*: Wenn Sie Ihr ladbares File im Overlay-Verfahren erstellen, ist das primäre Input-File das Basis-File, die nachgeladenen Files bilden den Rest der Programmstruktur.

3. *Bibliotheks-Files*: Diese beziehen sich auf benötigten Code aus Bibliotheken, die der Linker automatisch einbindet. Dabei können sowohl im Betriebssystem vorhandene (*resident Libraries*) als auch als Bibliothek erstellte Files (*scanned Libraries*) eingebunden werden. Der Linker lädt eine solche Bibliothek nur, wenn ein externes Symbol aus der Bibliothek benötigt wird.

Der Linker arbeitet ein File in zwei Durchläufen (*Passes*) ab.

1. Im ersten Durchlauf werden alle Primären, Bibliotheks- und Overlay-Files eingelesen sowie Code-Segmente und externe Symboltabellen definiert. Am Ende des ersten Durchlaufes werden, wenn gewünscht, die Cross-Referenz-Tabelle und die Link-Map erstellt.

2. Haben Sie ein Ausgabe-File angegeben, macht der Linker nun einen zweiten Durchlauf durch die Eingabe-Files. Zuerst wird das Primär-File in das Ausgabe-File kopiert. Dabei werden alle Referenzen aufgelöst. Dann werden auf gleichem Weg die Bibliotheks-Files eingebunden. Beachten Sie bitte, daß Bibliotheks-Routinen in den *Wurzel*-Teil des Overlays eingehen. Dann werden die Daten für den Overlay-Supervisor erstellt und die Overlay-Files in das Ausgabe-File kopiert.

Im ersten Durchlauf, nachdem alle Primär- und Overlay-Files eingelesen sind, überprüft der Linker die Symboltabellen auf noch nicht zugeordnete Symbole. Findet er solche, liest er die von Ihnen angegebenen Bibliothek-Files. Der Linker markiert dann alle Code-Segmente, die Definitionen für diese nicht aufgelösten Symbolzuweisungen enthalten. Der Linker bindet nur die Bibliotheks-Code-Segmente ein, auf die sich die Symboltabelle bezieht.

8.2 Zur Anwendung des Linkers

Um den Linker einzusetzen, müssen Sie die Befehls-Syntax, seine Ein- und Ausgaben und die möglicherweise auftretenden Fehler kennen. Dieser Abschnitt vermittelt Ihnen diese Kenntnisse.

8.2.1 Syntax der Befehlszeile

Der Aufruf von ALINK hat folgende Form:

```
ALINK [FORM|ROOT] <files> [TO <file>] [WITH <file>]
[VER <file>] [LIBRARY|LIB <files>] [MAP <file>]
[XREF <file>] [WIDTH n]
```

Das Muster für die Schlüsselworte ist:

```
"FROM=ROOT,TO/K,WITH/K,VER/K,LIBRARY=LIB/K,MAP/K,XREF/K,WIDTH/K"
```

Dabei steht <file> jeweils für ein File, <files> für eine Liste von null oder mehr Files und n für eine Integer-Zahl. File-Namen in Listen müssen durch Kommata oder »+« getrennt werden.

Folgende Zeilen sind beispielsweise gültige Aufrufe von ALINK:

```
ALINK a
ALINK ROOT a+b+c+d MAP mapfile WIDTH 120
ALINK a,b,c TO ausgabe LIBRARY :flib/lib,obj/neulib
```

Die Liste der Files kopiert der Linker in der Reihenfolge wie angegeben in das Ausgabe-File. Die einzelnen Parameter haben folgende Bedeutung:

FROM: Gibt die Objekt-Files an, die primär eingelesen werden. Der Linker kopiert diese Files zu dem Lade-File, um die Overlay-Wurzel zu erstellen. Ein primäres File muß angegeben werden. ROOT kann synonym zum FROM verwendet werden.

TO: Gibt das Ausgabe-File an. Wird dieses File nicht angegeben, wird kein zweiter Linker-Durchlauf gestartet.

WITH: Bezeichnet Files, in denen die Parameter für den Linker gespeichert sind. Normalerweise wird hier nur ein File eingegeben, jedoch können auch mehrere, sich ergänzende Files angegeben werden. Parameter in der Befehlszeile setzen die Parameter in »files« außer Kraft. Die genaue Form dieser Files wird in Abschnitt 8.2.2 dieses Handbuches geschildert.

VER: Bezeichnet das File, in das der Linker seine Meldungen schreibt. Ist dieses File nicht angegeben, wird die Ausgabe zum normalen Ausgabe-Device – im allgemeinen zum Bildschirm – geleitet.

LIBRARY: Bezeichnet die Files, die als Bibliotheken zur Auflösung externer Referenzen verwendet werden. Der Linker bindet nur Segmente ein, auf die mit einem externen Symbol hingewiesen wird. LIB ist eine gültige Abkürzung für LIBRARY.

MAP: Gibt das Ziel-File für die Link-Map an.

XREF: Gibt das Ziel-File für die Cross-Referenz-Tabelle an.

WIDTH: Definiert die Anzahl Zeichen pro Zeile, die für die Ausgabe der Link-Map und der Cross-Referenz-Tabelle verwendet werden. Dieser Parameter ist wichtig, wenn diese Ausgaben auf den Drucker geschrieben werden sollen.

8.2.2 WITH-Files

WITH-Files enthalten Parameter für den Linker. Sie werden diesen Befehl nur verwenden, wenn eine lange Sequenz von ALINK-Parametern häufiger verwendet werden soll oder die Namen der Files sehr *Tippfehler-Teufel-anfällig* sind.

Ein WITH-File besteht aus einer Reihe von ALINK-Schlüsselwörtern, je einem pro Zeile, mit den dazugehörigen Argumenten. Eine Zeile kann mit einem Semikolon abgeschlossen werden, der Linker ignoriert dann den Rest der Zeile. Nach dem Semikolon können Sie

Kommentare eingeben. Leerzeilen werden ignoriert. An Schlüsselwörtern stehen zur Verfügung:

```
FROM/ROOT files
TO file
LIBRARY files
MAP [file]
XREF [file]
OVERLAY Struktur-Angabe
#
WIDTH zahl
```

Dabei steht »file« für ein einzelnes File, »files« für eine Liste von einem oder mehreren Files und [file] für ein optionales File. »zahl« muß eine ganze Zahl sein. Der Stern (*) kann zum Aufteilen langer Zeilen benutzt werden: Ein Stern am Ende einer Zeile teilt dem Drucker mit, daß die folgende Zeile zu der mit dem Stern gehört. Geben Sie nach MAP oder XREF kein File an, werden die entsprechenden Tabellen zum File VER geschrieben und dadurch normalerweise auf den Bildschirm.

Parameter, die Sie in der Befehlszeile eingeben, ersetzen Parameter zum gleichen Schlüsselwort in WITH-Files. So können Sie Standard-WITH-Files mit abweichenden Parametern nutzen. Geben Sie innerhalb des WITH-Files mehrere Parameter für ein Schlüsselwort an, wird das erste eingesetzt.

Auch in dem zweiten untenstehenden Beispiel trifft das übrigens zu, selbst wenn einem Schlüsselwort der Parameter Null zugewiesen wird.

Hier nun ein paar Beispiele zu WITH-Files und den dazugehörigen ALINK-Aufrufen:

```
ALINK WITH link-file
```

wobei »link-file« so aussieht:

```
FROM     obj/haupt,obj/nächst
TO       bin/test
LIBRARY  obj/lib
MAP
XREF     x0
```

erzielt das gleiche Ergebnis wie:

```
ALINK FROM obj/haupt,obj/nächst TO bin/test LIBRARY obj/lib MAP XREF x0
```

Der Befehl

```
ALINK WITH lkin LIBRARY ""
```

wobei »lkin« enthält:

```
FROM bin/prog,bin/subs
LIBRARY nag/fort.lib
TO linklib/prog
```

gibt das gleiche Ergebnis wie der Befehl:

```
ALINK FROM bin/prog,bin/subs TO linklib/prog
```

In dem obigen Beispiel ersetzt der Parameter ohne Wert in der Befehlszeile den Parameter im Link-File. Deshalb wird kein Bibliotheks-File eingebunden.

8.2.3 Fehler und andere Ausnahmebehandlungen

Während des Linker-Durchlaufes können verschiedene Fehler auftreten. Die meisten Meldungen erklären sich selbst und verweisen auf Fehler beim Öffnen der Files oder auf Fehler innerhalb der Parameter oder eines Binär-Files. Ein Fehler beendet meist den Link-Vorgang.

Einige Fehler ergeben jedoch nur eine Warnung. Die wichtigsten verweisen auf undefinierte oder mehrmals definierte Symbole. Der Linker-Durchlauf wird nach einer Warnung nicht abgebrochen.

Wird im ersten Durchlauf ein undefiniertes Symbol entdeckt, erzeugt der Linker eine Warnung und gibt eine Liste solcher Symbole aus. Während des zweiten Durchlaufes verweisen nicht definierte Symbole immer zur Adresse Null.

Findet der Linker während des ersten Durchlaufes mehrere Definitionen für ein Symbol, so erzeugt er eine Warnung und ignoriert die zweite und jede weitere Definition. Diese Meldung wird nicht ausgegeben, wenn die Definition in einem LIBRARY-File auftaucht. Sie können also LIB-Funktionen einbinden, ohne einen Fehler zu erzeugen, auch wenn im Quell-Code und in einem LIB-File das gleiche Symbol verwendet wird. Ein ernsthafter Fehler tritt allerdings auf, wenn der Linker auf widersprüchliche Symbol-Zuweisungen stößt. Der Linkvorgang wird sofort abgebrochen.

Da der Linker nur die erste Definition eines Symbols übernimmt, ist es wichtig, daß Sie berücksichtigen, in welcher Reihenfolge Files eingelesen werden.

1. Primäre Eingabe-Files (FROM oder ROOT).
2. Overlay-Files.
3. LIBRARY-Files.

Innerhalb jeder Gruppe werden die Files in der im Argument angegebenen Reihenfolge eingelesen. Also haben Definitionen in FROM-Files die höchste und solche in LIBRARY-Files die niedrigste Priorität.

8.2.4 MAP und XREF

Die Link-Map, die der Linker am Ende des ersten Durchlaufes erzeugt, listet alle Code-Segmente, die im zweiten Durchlauf in dieser Reihenfolge in das ladbare File geschrieben werden.

Für jedes Code-Segment gibt der Linker eine Kopf-Zeile aus, bestehend aus dem Namen des Files, gekürzt auf acht Buchstaben, der Code-Segment-Referenznummer, dem Datentyp (Data, Code, BSS oder COMMON) und der Größe. Ist es ein Overlay-File, werden dazu noch die Overlay-Ebene und der Overlay-Rang ausgegeben.

Nach dem Kopf gibt der Linker jedes in dem Code-Segment definierte Symbol mit seinem Wert aus. Die Symbole werden nach ihrem Wert geordnet ausgegeben. Absolut definierten Symbolen wird ein »*« angehängt.

Der Wert des WIDTH-Schlüsselwortes gibt die Anzahl an Zeichen pro Zeile für diese Tabelle an. Ist der Wert sehr klein eingestellt, wird ein Symbol pro Zeile ausgegeben.

Die Cross-Referenz-Tabelle listet jedes Code-Segment. Die Kopfzeile ist dabei die gleiche wie bei MAP.

Der Kopfzeile folgt eine Liste der Symbole mit ihren Referenzen. Jede Referenz enthält zwei Integer-Zahlen, die den Versatz der Referenz und die Nummer des Code-Segmentes, auf das die Referenz weist, anzeigen. Die Nummer eines Code-Segmentes steht in seiner Kopfzeile.

8.3 Overlays

Das Overlay-System des Linkers und der Overlay-Supervisor erlauben es, große Programme abzuarbeiten, ohne zu viel Speicherplatz zu belegen. Eingriffe in den Objekt-Code sind dazu nicht nötig.

Wenn ein Programm im Overlay-Modus abgearbeitet wird, können Sie sich dessen Struktur wie einen Baum vorstellen. Die *Wurzel* des Baumes ist das Primärfile, zusammen mit den LIB-Segmenten und den COMMON-Blocks. Diese Wurzel ist immer im Hauptspeicher vorhanden. Die Overlay-Files bilden die *Äste* unseres Baumes. Sie sind angeordnet, wie in der Overlay-Direktive angegeben.

Das vom Linker erzeugte Lade-File besteht auch beim Overlay-Modus aus einem Binär-File, das alle Code-Segmente und die *Einbau-Anleitung* für die Äste des Overlay-Baumes enthält. Laden Sie nun das Programm, wird zuerst nur die Wurzel in den Speicher geladen. Der

Overlay-Supervisor übernimmt dann das Laden und Löschen der Äste des Programmes. Dieser Overlay-Supervisor wird beim Link-Vorgang automatisch mit eingebunden, wenn Overlays benutzt werden. Die Arbeit des Overlay-Supervisors ist für den Anwender und Programmierer nicht bemerkbar.

8.3.1 Die OVERLAY-Direktive

Um die Struktur des Programm-»Baumes« zu definieren, wird die Direktive OVERLAY eingesetzt. Diese Direktive darf nur in WITH-Files angewandt werden. Die erste in einem WITH-File angegebene OVERLAY-Direktive wird ausgeführt.

Das Format für OVERLAY lautet:

```
OVERLAY
xfiles
.
.
.
#
```

Die Overlay-Direktive kann sich über mehrere Zeilen erstrecken. Das Ende der Direktive wird mit »#« angegeben. Findet der Linker dieses Zeichen nicht, verwendet er alle Zeilen bis zur End-of-File-Marke als Argumente der OVERLAY-Direktive.

Jede Zeile nach OVERLAY definiert einen *Zweig des Programm-Baumes* und besteht aus einem Zeichen »X« und einer Liste von Files.

Die Ebene eines Astes in der *Höhe* des Baumes wird mit keinem, einem oder mehreren »*« definiert. Wird kein Stern eingegeben, gehört des Overlay-File zur Ebene der Wurzel. Die Overlay-Ebene eines Files ist x+1. Zusätzlich zur Ebene hat jedes File noch einen Ordnungswert. Er gibt die Reihenfolge an, in der die Files eines Astes gelesen werden sollen. File 1 ist das erste File nach der Verzweigung. Es kann mehrere Äste mit gleicher Ebene und gleichem Ordnungswert geben. Sie müssen jedoch von verschiedenen Ursprungsästen ausgehen!

Während der Linker eine Overlay-Direktive bearbeitet, erinnert er sich der aktuellen Ebene und vergleicht die angegebene Höhe für jeden neuen Ast mit der vorhergehenden Nummer. Ist die neue Zahl kleiner, ist der neue Ast ein *Abkömmling* eines vorangehenden. Ist sie gleich, hat der neue Ast denselben Ursprung wie der letzte Ast. Ist die Zahl größer, ist der neue Ast ein direkter Abkömmling des letzten.

Hier einige Beispiele zur Verdeutlichung:

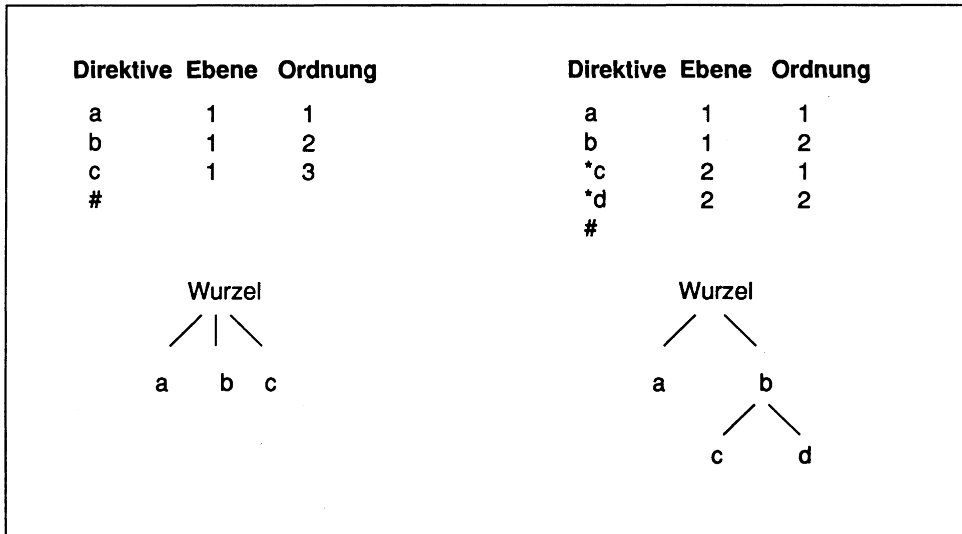


***Bild 8.2:** Unterschiedliche OVERLAY-Bäume*

Während des Link-Vorganges werden die Overlay-Files, die Sie in der Direktive angegeben haben, Zeile für Zeile eingelesen. Die Reihenfolge wird in der Link-Map und der Cross-Reference-Tabelle festgehalten. Die Overlay-Ebenen und Ordnungszahlen werden dabei mit ausgegeben, so daß Sie die Struktur des Programmes jederzeit rückverfolgen können.

Die Ebenen und Ordnungszahlen beziehen sich ja auf den Ursprung in derselben Zeile. Bitte beachten Sie, daß alle oben mit den einzelnen Buchstaben bezeichneten Files auch ganze Listen von Files sein können.

Zum Beispiel:

```
ROOT bin/hauptaaa
OVERLAY
bin/hauptbbb,bin/hauptccc,bin/hauptddd
*bin/machreal
*bin/trbblock,bin/transint,bin/transr*
bin/transri
bin/codeaus
#
```

ergibt den im folgenden Bild gezeigten Programmbaum:

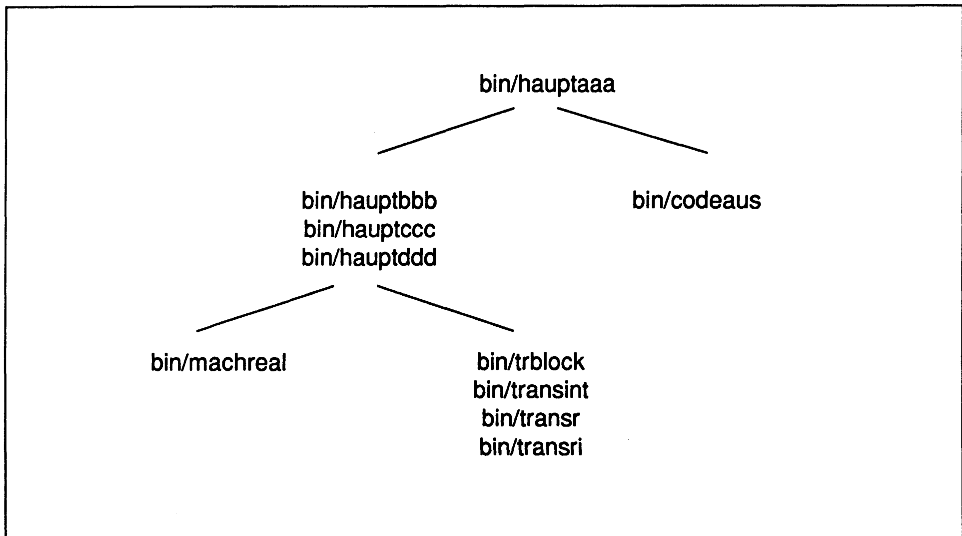


***Bild 8.3:** Ein Programmbaum*

8.3.2 Symbol-Bezüge

Während ein Programm mit Overlay gelinkt wird, überprüft der Linker jeden Symbol-Bezug auf Zulässigkeit.

Nehmen wir an, daß der Bezug in der Verzweigung »R« und das Symbol in der Verzweigung »S« steht. Die Zuweisung ist zulässig, wenn eine der drei untenstehenden Bedingungen wahr ist.

1. »R« und »S« stehen in derselben Verzweigung.
2. »R« steht in einer nachgeordneten Ebene zu »S«.
3. »R« und »S« stehen in zwei verschiedenen Zweigen, aber auf gleicher Ebene.

Bezugnahmen der dritten Regel werden auch Overlay-Bezüge genannt. In diesem Fall startet der Linker den Overlay-Supervisor während des Programmablaufes. Der Supervisor überprüft dann, ob das Segment mit dem Symbol noch im Speicher steht. Wenn nicht, werden alle vorhandenen Segmente dieser Ebene und tiefer gelöscht und dann das mit dem Symbol eingeladen. Ein Overlay-Segment springt direkt zum aufrufenden Segment zurück, daher wird es nicht gelöscht, bis ein Segment höherer Ebene »darüber«geladen wird.

Angenommen, der Programm-Baum sieht wie folgt aus:

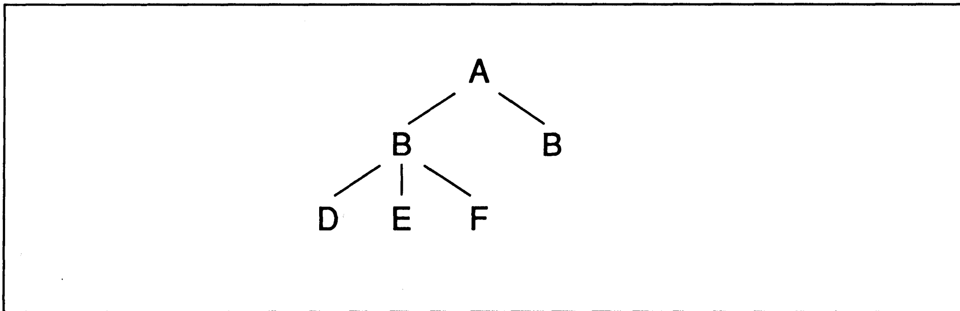


Bild 8.4: Ein einfacher Programmbaum

Lädt der Linker das Programm, steht zunächst nur Teil A im Speicher. Findet der Linker in A ein Symbol mit Bezug auf B, wird dieses Segment geladen und gestartet. Wenn B nun D aufruft, wird eine neue Verzweigung geladen. Verweist B auf A zurück, verbleiben die Segmente B und D im Speicher und brauchen nicht neu geladen zu werden, wenn auf sie verwiesen wird. Nehmen wir nun an, A ruft C auf. Nun löscht der Linker erst die nicht mehr benötigten Segmente und macht damit den Speicher für den neuen Teil frei. In diesem Fall also B und D. Erst wenn diese Teile gelöscht sind, kann der Linker C laden.

Wird ein Segment ausgeführt, sind alle darüberliegenden Segmente und eventuell einige nachgeordnete Segmente im Speicher.

8.3.3 Zusätzliche Hinweise

Der Linker nimmt an, daß Referenzen in andere Teile des Overlays Sprünge oder Unterprogramm-Aufrufe sind. Diese Verzweigungen werden mit dem Overlay-Supervisor gesteuert. Sie sollten deshalb keine Overlay-Symbole für den Zugriff auf Daten verwenden.

Der Linker erteilt jedem Segment, das eine Overlay-Zuweisung enthält, eine Overlay-Nummer. Diese Nummer, die immer größer oder gleich Null ist, wird benutzt, um ein Label zu konstruieren, das dem Einsprungpunkt in dieses Overlay entspricht. Das Label hat die Form »OVLYnnn« wobei »nnn« die eben erwähnte Overlay-Nummer ist. Dieses Format für Labels sollten Sie deshalb nicht selbst verwenden.

Der Linker zieht alle Sektionen mit demselben Sektions-Namen zu einem Code-Segment zusammen. Damit können diese Segmente einfacher in einem Stück in den Speicher eingeladen werden.

8.4 Fehlercodes und -meldungen

Diese Fehler werden sehr selten auftreten. Wenn doch, kann der Fehler auch am Compiler liegen. Sie sollten trotzdem überprüfen, ob Ihre Programme einwandfrei sind (z.B. muß ein Eingabe-Programm einen Einführungsteil enthalten, das dem Linker mitteilt, daß ein Programm zu erwarten ist).

Unzulässige Objektmodule

2 Overlay-Symbol falsch angewandt
3 Symbol falsch angewandt
4 COMMON falsch angewandt
5 Overlay-Zuweisung nicht korrekt
6 Overlay-Zuweisung nicht Null
7 Externen Block falsch verschoben
8 bss falsch verschoben
9 Programmteil falsch verschoben
10 Falscher Offset bei 32-Bit-Verschiebung
11 Falscher Offset bei 6/8-Bit-Verschiebung
12 Falscher Offset bei 32-Bit Zuweisung
13 Falscher Offset bei 6/8-Bit-Zuweisung
14 End-of-file nicht gefunden
15 Hunk.end nicht gefunden
16 Unzulässiges Fileende
17 File vorzeitig beendet
18 File vorzeitig beendet

Interne Fehler

19 Unzulässiger Typ in Hunk-Liste
20 Interner Fehler beim Libraryaufruf
21 Falsches Argument zu Free-Vektor
22 Symbol im zweiten Durchlauf nicht definiert

Anhang: Terminal-Ein- und -Ausgabe auf dem Amiga

In diesem Anhang stehen die Zeichen <CSI> für *Control Sequence Introducer*. Zur Ausgabe können Sie <Esc>-[oder $9B (hex) verwenden, bei der Eingabe wird $9B gemeldet.

Einführung

Dieser Anhang beschreibt verschiedene Wege, um das Terminal (Tastatur und Bildschirm) für Ein- und Ausgaben zu verwenden. Dieses Terminal kann wie ein normales AmigaDOS-File (mit *, CON: und RAW:) oder direkt durch Aufruf der CONSOLE.LIBRARY angesprochen werden. Hier nun die Vor- und Nachteile der verschiedenen Methoden:

* Der Stern erzeugt kein neues Fenster, sondern nutzt das aktive Fenster des CLI. Nicht der gesamte Zeichensatz steht zur Verfügung. Möglich sind Groß- und Kleinbuchstaben (a–z und A–Z), ASCII-Symbole und Control-Zeichen. Jedes Zeichen, das ein Fernschreiber mit einer Taste erzeugen kann, wird ebenfalls dargestellt. Zusätzlich können alle anderen Zeichen verwendet werden, bei denen das höchste Bit gesetzt ist. ($80–$FF). Die Zeichen <Backspace> und <Ctrl>-X zum Löschen eines Zeichens beziehungsweise einer Zeile werden ebenfalls unterstützt. Jede <CSI>-Sequenz kann die <Ctrl>-Zeichen C, D, E, F, H und X enthalten. Jedes <CR> oder <Ctrl>-M wird in <Ctrl>-J (neue Zeile) umgewandelt.

CON: Wie »*«, es wird allerdings ein neues Fenster definiert.

RAW: Der *einfache Fall*: Mit RAW: (verglichen mit CON:) verlieren Sie die Möglichkeit, den Text in der Zeile noch zu verändern. Davon sind die Cursor-Tasten ebenso betroffen wie die Ctrl-Tasten. Alle Steuerzeichen werden als Zeichen übernommen. Der *komplizierte Fall*: Bei Eingabe

zusätzlicher Befehle an den Console-Prozessor können Sie weit mehr Informationen erhalten. Zum Beispiel können Sie Informationen über einen Tastendruck oder Maus-Klicks erhalten. Weitere Informationen dazu im Abschnitt *Auswahl von RAW:-Ereignissen* später in diesem Anhang.

console.device: Mit diesem Verfahren haben Sie vollen Zugriff auf alle Möglichkeiten des Terminals. Sie können die Tastenbelegung ändern und zum Beispiel auch das komplette Keyboard umdefinieren.

Hilfreiche AmigaDOS-Befehle

Zwei sehr hilfreiche AmigaDOS-Befehle sollen Sie zu eigenen Experimenten anregen:

```
TYPE RAW:10/10/100/30/ opt h
```

übernimmt Eingaben aus einem neuen RAW:-Fenster und zeigt die ASCII-Werte der gedrückten Tasten als Hexadezimalzahlen an. Dies ist der einfachste Weg, um zu überprüfen, welche Zeichen von der Tastatur gesendet werden.

Und der Befehl

```
COPY "RAW:10/10/100/30/RAW Eingabe" "RAW:100/10/200/100/RAW Ausgabe"
```

erlaubt, Zeichen in das (nicht aktive) Eingabefenster zu schreiben und die Zeichen im (aktiven) Ausgabefenster darzustellen. COPY erkennt die End-of-File-Marke der RAW:-Eingabe nicht. Sie müssen das System nach diesem Befehl neu starten.

CON:-Tastatur-Eingabe

Lesen Sie Daten vom CON:-Device, werden die Eingaben von der Tastatur zwischengespeichert. Sie erhalten ASCII-Zeichen wie »B«. Die meisten Programme, die Eingaben von der Tastatur erwarten, lesen aus CON:. Einige, zum Beispiel Textverarbeitungs-Programme oder Programme, die die Tastatur als *Klaviatur* verwenden, lesen jedoch aus RAW:.

Um die Sonderzeichen und internationalen Zeichen zu erhalten, drücken Sie zusätzlich die ALT-Taste. Sie setzt das High-Bit des ASCII-Codes der gedrückten Taste.

Der Code $FF (Umlaut-y) ist ein besonderer Fall. Normalerweise müßte <Alt>-<Del> zu diesem Code führen. Der ASCII-Code <Del> (hex $7F) ist jedoch kein druckbares Zeichen. Da es unseren Vorstellungen widerspricht, daß ein druckbares Zeichen von einem nicht druckbaren erzeugt wird, wurde der normale ALT-Modus in diesem Fall geändert.

In Tabelle A.1 werden alle Zeichen gezeigt, die auf dem Amiga dargestellt weren können. Die Zeichen NBSP (nicht trennendes Leerzeichen) und SHY (optionaler Trennstrich) werden nur in Textverarbeitungs-Programmen eingesetzt.

	01	02	03	04	05	06	07	08	09	10	10	11	12	13	14	15
00			sp	0	@	P	`	p			nbsp	°	À	D	à	δ
01			!	1	A	Q	a	q			¡		Á	Ñ	á	ñ
02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
04			$	4	D	T	d	t			♦	´	Ä	Ô	ä	ô
05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
06			&	6	F	V	f	v			|	¶	Æ	Ö	æ	ö
07			'	7	G	W	g	w			§	•	Ç	□	ç	□
08			(	8	H	X	h	x			¨	,	È	Ø	è	ø
09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù
10			*	:	J	Z	j	z			ª	º	Ë	Ú	ê	ú
11			+	;	K	[	k	{			«	»	E	Û	ë	û
12			,	<	L	\	l	|			¬	a	Ì	Ü	ì	ü
13			-	=	M	]	m	}			shy	a	Í	Y	í	y
14			.	>	N	^	n	~			®	a	Î	Þ	î	þ
15			/	?	O	_	o				—	¿	Ï	ß	ï	ÿ

***Tabelle A.1:** Internationaler Zeichencode*

AmigaDOS verwendet CON: als Eingabe für das CLI und die meisten anderen CLI-Befehle. Dieses Device filtert jedoch alle Funktionstasten und die Cursortasten aus. Programme unter AmigaDOS können (und viele tun das auch) das Device RAW: öffnen. Damit sind Eingaben über die Funktions- und Cursortasten möglich.

CON:-Bildschirm-Ausgabe

Die Bildschirm-Ausgabe mit CON: ist ähnlich der mit RAW:, jedoch wird das Zeichen <LF> (hex $0A) in das Zeichen für *neue Zeile* (CR) umgewandelt. Der Cursor springt also immer in die erste Spalte einer neuen Zeile, wenn das Zeichen <LF> erscheint.

RAW:-Bildschirm-Ausgabe

Die im Standard ANSI x3.64 definierten Codes werden unterstützt:

Unabhängige Kontroll-Codes (keine anderen Tasten werden vorher betätigt):

<Ctrl>	Hex	Name	Definition	Beschreibung
H	08	BS	BACKSPACE	Bewegt den Cursor ein Zeichen nach links
I	09	TAB	TAB	Bewegt den Cursor ein Zeichen nach rechts
J	0A	LF	LINE FEED	Zeilenvorschub
K	0B	VT	VERTICAL TAB	Bewegt den Cursor eine Zeile nach unten und scrollt wenn nötig
L	0C	FF	FORM FEED	Löscht den Bildschirm
M	0D	CR	CARRIAGE RETURN	Springt zur ersten Spalte
N	0E	SO	SHIFT OUT	Setzt MSB für jedes Zeichen vor Ausgabe
O	OF	SI	SHIFT IN	Widerruft SHIFT OUT
[	1B	Esc	ESCAPE	Siehe unten

Vor den folgenden Zeichen muß <Esc> betätigt werden:

Zeichen	Name	Definition	Beschreibung
c	RIS	RESET TO INITIAL STATE	Reset des Systems

Vor den folgenden Zeichen muß <Esc> oder die Tastenkombination <Ctrl>-<Alt> und der angegebene Buchstabe gedrückt werden, um die Aktion auszuführen:

Zeichen	Hex	Name	Beschreibung
IND	845tD	INDEX:	Eine Zeile nach unten
E	85	NEL	Nächste Zeile
M	8D	RI	Eine Zeile nach oben
[	9B	CSI	CONTROL SEQUENCE INTRODUCER: siehe nächste Liste

Es folgen nun Kontroll-Sequenzen (mit vorangestelltem <CSI>) mit Parametern. Das erste Zeichen in der folgenden Tabelle (unter dem Zeichen <CSI>) zeigt die Anzahl der erlaubten Parameter an. Dabei gilt:

»0« keine Parameter erlaubt
»1« kein oder ein numerischer Parameter
»2« zwei numerische Parameter ("14;94")
»3« eine beliebige Anzahl numerischer Parameter, getrennt durch ein Semicolon (;)
»4« genau 4 numerische Parameter
»8« genau 8 numerische Parameter

<CSI>	**Name**	**Definition**	**Beschreibung**
1 @	ICH	INSERT CHARACTER	Setzt ein oder mehrere Leerzeichen ein, der Rest der Zeile wird nach rechts verschoben
1 A	CUU	CURSOR UP	Cursor nach oben
1 B	CUD	CURSOR DOWN	Cursor nach unten
1 C	CUF	CURSOR FORWARD	Cursor nach rechts
1 D	CUB	CURSOR BACKWARD	Cursor nach links
1 E	CNL	CURSOR NEXT LINE	n Zeilen abwärts in die 1. Spalte
1 F	CPL	CURSOR PRECEDING LINE	n Zeilen aufwärts in die 1. Spalte
2 H	CUP	CURSOR POSITION	»<CSI>Zeile,SpalteH«
1 J	ED	ERASE IN DISPLAY	Lösche auf dem Bildschirm (nur zum Ende der Ausgabe)
1 K	EL	ERASE IN LINE	Lösche in der Zeile (nur zum EOL)
1 L	IL	INSERT LINE	Fügt eine Zeile vor der Zeile mit dem Cursor ein
1 M	DL	DELETE LINE	Löscht die aktuelle Zeile. Bewegt alle nachfolgenden Zeilen um eine nach oben, löscht die letzte Zeile
1 P	DCH	DELETE CHARACTER	Löscht Zeichen
2 R	CPR	CURSOR POSITION REPORT	(nur im Lese-Strom)Format des Reports: »<CSI> Zeile; SpalteR«
1 S	SU	SCROLL UP	Entfernt erste Zeile vom Bildschirm, bewegt alle anderen eine Zeile nach oben, löscht letzte Zeile
1 T	SD	SCROLL DOWN	Entfernt letzte Zeile auf dem Bildschirm, bewegt alle anderen eine Zeile nach unten, löscht erste Zeile
3 h	SM	SET MODE	<CSI>2Oh weist RAW: an, <LF> als »neue Zeile« zu interpretieren.

3 1	RM	RESET MODE	<CSI>201 widerruft SET MODE 20
3 m	SGR	SELECT GRAPHIC RENDITION	Schaltet Grafik-Modus ein.
1 n	DSR	DEVICE STATUS REPORT	

Die folgenden Steuercodes sind keine ANSI-Standard-Sequenzen, sondern werden nur im Amiga verwendet:

<table>
<tr><th><CSI></th><th>Name</th><th>Definition</th><th>Beschreibung</th></tr>
<tr><td>1 t</td><td>aSLPP</td><td>SET PAGE LENGTH</td><td>Stellt Papierlänge ein</td></tr>
<tr><td>1 u</td><td>aSLL</td><td>SET LINE LENGTH</td><td>Stellt Zeilenlänge ein</td></tr>
<tr><td>1 x</td><td>aSLO</td><td>SET LEFT OFFSET</td><td>Setzt linken Versatz</td></tr>
<tr><td>1 y</td><td>aSTO</td><td>SET TOP OFFSET</td><td>Setzt oberen Versatz</td></tr>
<tr><td>3 {</td><td>aSRE</td><td>SET RAW EVENTS</td><td>Setzt RAW:-Ereignisse</td></tr>
<tr><td>8 |</td><td>aIER</td><td>INPUT EVENT REPORT</td><td>Report über Eingabe-Ereignisse (bei Lese-Strom)</td></tr>
<tr><td>3 }</td><td>aRRE</td><td>RESET RAW EVENTS</td><td>Rücksetzen der RAW-Ereignisse</td></tr>
<tr><td>1 ~</td><td>aSKR</td><td>SPECIAL KEY REPORT</td><td>Report über Sondertasten (bei Lese-Strom)</td></tr>
<tr><td>1 p</td><td>aSCR</td><td>SET CURSOR RENDITION</td><td>Schaltet Cursor ein,
<Esc> p schaltet den Cursor aus</td></tr>
<tr><td>0 q</td><td>aWSR</td><td>WINDOW STATUS REQUEST</td><td>Fenster-Status ausgeben</td></tr>
<tr><td>4 r</td><td>aWBR</td><td>WINDOW BOUNDS REPORT</td><td>Größe eines Fensters (bei Lese-Strom)</td></tr>
</table>

Beispiele:

Cursor ein Zeichen nach rechts setzen:

<CSI>C oder <TAB> oder <CSI>1C

Cursor 20 Zeichen nach rechts setzen:

<CSI>20C

Cursor in die linke obere Ecke des Bildschirms (*Home Position*) setzen:

<CSI>H oder <CSI>1;1H oder <CSI>1;H

Cursor in die vierte Spalte der ersten Zeile des Fensters setzen:

<CSI>1;4H oder <CSI>;4H

Bildschirm löschen:

<FF> oder <Ctrl>-L	;Zeichen »Clear Screen« oder
<CSI>H<CSI>J	;»Home« und Rest des Bildschirms löschen oder
<CSI>H<CSI>23M	;»Home« und 23 Zeilen löschen oder
<CSI>1;1H<CSI>23L	;»Home« und 23 Zeilen einfügen

RAW:-Tastatur-Eingabe

Lese-Zugriffe auf ein RAW:-File ergeben einen Byte-Strom nach dem Standard ANSI x3.64. Dieser Strom enthält normale Zeichen und *RAW:-Ereignis-Informationen.* Diese können mit den SET RAW EVENTS (aSRE) und RESET RAW EVENTS (aRRE) Kontroll-Sequenzen angefordert werden, wie unten beschrieben.

Haben Sie eine RAW:-Eingabe angefordert, wenn kein Datenfluß stattfindet, wartet der Rechner, bis wieder Daten zur Verfügung gestellt werden. Auf Dateneingang kann mit der Funktion »WaitForChar« getestet werden.

Bei normaler Einstellung werden von den Funktionstasten und Cursortasten folgende Zeichen zurückgeliefert:

Taste	**ohne Shift**	**mit Shift**	
F1	<CSI>0~	<CSI>10~	
F2	<CSI>1~	<CSI>11~	
F3	<CSI>2~	<CSI>12~	
F4	<CSI>3~	<CSI>13~	
F5	<CSI>4~	<CSI>14~	
F6	<CSI>5~	<CSI>15~	
F7	<CSI>6~	<CSI>16~	
F8	<CSI>7~	<CSI>17~	
F9	<CSI>8~	<CSI>18~	
F10	<CSI>9~	<CSI>19~	
HELP	<CSI>?~	<CSI>?~	(identisch mit und ohne Shift)

Cursortasten:

Taste	**ohne Shift**	**mit Shift**	
Auf	<CSI>A	<CSI>T~	
Ab	<CSI>B	<CSI>S~	
Links	<CSI>C	<CSI> A~	(beachten Sie das Leerzeichen)
Rechts	<CSI>D	<CSI> @~	(beachten Sie das Leerzeichen)

Auswahl von RAW:-Ereignissen

Nutzen Sie RAW:, erhalten Sie die ANSI-Daten und die gerade beschriebenen zusätzlichen Kontroll-Codes. Brauchen Sie mehr Informationen über bestimmte Eingabe-Ereignisse, können Sie diese vom Console Driver anfordern.

Müssen Sie zum Beispiel wissen, wann welche Taste gedrückt und wieder losgelassen wurde, müssen Sie RAW:-Tastatur-Eingabe verlangen. Das geschieht mit Ausgabe von <CSI>1{ an das Terminal.

RAW:-Eingabe-Ereignisse:

Zugriffs-Nummer	**Beschreibung**
0	Keine Operation; wird intern genutzt
1	RAW:-Tastatur-Eingabe
2	RAW:-Maus-Eingabe
3	Ereignis; wird immer dann gesendet, wenn ein Fenster aktiviert wird.
4	Pointer-Position
5	(unbenutzt)
6	Timer
7	Gadget gedrückt
8	Gadget losgelassen
9	Requester aktiv
10	Menü-Nummer
11	Schließ-Gadget
12	Fenstergröße verändert
13	Fenster neu gezeichnet
14	Preferences verändert
15	Disk entnommen
16	Disk eingelegt

Wählen Sie eines dieser Ereignisse an, erhalten Sie weitere Informationen in der folgenden Form:

<CSI><Gruppe>
<Untergruppe>
<Tastencode>
<Indikator>
<X>
<Y>
<Sekunden>
<Microsekunden>

<CSI> ist ein Ein-Byte-Feld. Es ist immer gleich $9B hexadezimal.

<Gruppe> ist der RAW:-Eingabe-Ereignis-Code aus obenstehender Tabelle.

<Untergruppe> ist nicht benutzt und immer gleich Null.

<Tastencode> zeigt an, welche Taste gedrückt wurde (Bild A.1 und Tabelle A.2 zeigen die Nummern). Mit diesem Feld wird auch die Maus abgefragt.

<Indikator> zeigt den Status von Tastatur und System. Dieses Feld ist wie folgt definiert:

Bit	**Maske**	**Taste**	
0	0001	Umstell-Taste links	
1	0002	Umstell-Taste rechts	
2	0004	Umstell-Taste-Feststeller	* Sondertaste, siehe unten
3	0008	Control	
4	0010	Alt-Taste links	
5	0020	Alt-Taste rechts	
6	0040	linke Amiga-Taste gedrückt	
7	0080	rechte Amiga-Taste gedrückt	
8	0100	10er-Block	
9	0200	Wiederholung	
10	0400	Interrupt; nicht benutzt	
11	0800	mehrfach ausgegeben; dieses (das aktive) oder alle Fenster	
12	1000	linker Maus-Knopf	
13	2000	rechter Maus-Knopf	
14	4000	mittlerer Maus-Knopf; mit der Standard-Maus nicht erreichbar	
15	8000	Maus relativ; angegebene Mauskoordinaten sind relativ und nicht absolut	

Die Taste zum Feststellen der Umschalt-Taste wird besonders behandelt. Sie liefert nur einen Tastencode, wenn sie gedrückt ist, nicht jedoch, wenn sie wieder losgelassen ist. Das entsprechende Bit, das Großbuchstaben signalisiert (80 hex), wird jedoch korrekt gemeldet. Drücken Sie die Feststelltaste, so daß die LED aufleuchtet, wird der Tastencode 62 (gedrückt) gesendet. Drücken Sie nun wieder die Taste und die Leuchtdiode geht aus, wird

190 (losgelassen) gesendet. Die Tastatur meldet diese Taste also so lange als gedrückt, bis sie ein zweites Mal benutzt wird.

Die Felder <Sekunden> und <Microsekunden> enthalten die Zeit, zu der das Ereignis stattfand, im Format der Systemzeit. Diese wird als Langwort gespeichert.

Mit der RAW:-Tastatur-Eingabe meldet die Taste nicht nur ein einzelnes Zeichen, sondern eine Menge mehr an Informationen im Format:

<CSI>1;0;<tastencode>;<indikator>;0;0;<sekunden>;<microsekunden>|

Hierzu ein Beispiel: Drücken Sie die Taste »B« und lassen Sie sie wieder los, während die linke Umstelltaste und die rechte Amiga-Taste gedrückt waren, erhalten Sie folgende Informationen über RAW:

<CSI>1;0;35;129;0;0;23987;99|
<CSI>1;0;163;129;0;0;24003;18|

Die Felder mit dem Inhalt »0;0« werden bei Tastatur-Eingaben nicht benutzt. Sie dienen zum Auslesen der Maus-Koordinaten und liefern die X- und Y-Koordinaten des Mauszeigers.

Das Feld <tastencode> enthält den Dezimalwert des ASCII-Codes der Taste, die gedrückt wurde. Zählen Sie 128 dazu, erhalten Sie den Code der losgelassenen Taste. Bild A.1 erleichtert Ihnen die Suche nach dem Tastencode einer bestimmten Taste, mit Bild A.2 finden Sie leicht eine Taste nach ihrem Tastencode.

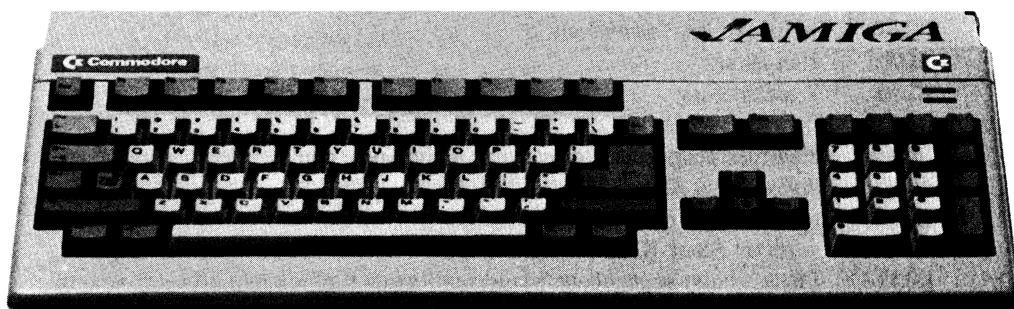


Foto Commodore

Bild A.1: *Vereinfachte Darstellung der Amiga-Tastatur-Belegung*

In der nachfolgenden Tabelle finden Sie:

1. Den Wert, den CON: meldet, wenn eine bestimmte Taste gedrückt wurde, und

2. Die Belegung der deutschen Standard-Tastatur.

RAW:- Tasten- Code	**Wert ohne Shift-Taste**	**Wert mit Shift-Taste**
00	[	]
01	1	!
02	2	"
03	3	#
04	4	$
05	5	%
06	6	&
07	7	/
08	8	(
09	9	)
0A	0	=
0B	ß	?
0C	´	`
0D	\	|
0E	nicht definiert	
0F	0	0 (Zehnerblock)
10	Q	q
11	W	w
12	E	e
13	R	r
14	T	t
15	Z	z
16	U	u
17	I	i
18	O	o
19	P	p
1A	Ü	ü
1B	+	*
1C	nicht definiert	
1D	1	1 (Zehnerblock)
1E	2	2 (Zehnerblock)
1F	3	3 (Zehnerblock)

Tabelle A.2: *Tastaturcodes (Fortsetzung nächste Seite)*

RAW:- Tasten- Code	**Wert ohne Shift-Taste**	**Wert mit Shift-Taste**
20	A	a
21	S	s
22	D	d
23	F	f
24	G	g
25	H	h
26	J	j
27	K	k
28	L	l
29	Ö	ö
2A	Ä	ä
2B	#	^
2C	nichtdefiniert	
2D	4	4 (Zehnerblock)
2E	5	5 (Zehnerblock)
2F	6	6 (Zehnerblock)
30	<	>
31	Y	y
32	X	x
33	C	c
34	V	v
35	B	b
36	N	n
37	M	m
38	, (Komma)	;
39	. (Punkt)	:
3A	– (Bindestrich)	_ (Unterstrich)
3B	nicht definiert	
3C	. (Punkt)	. (Zehnerblock)
3D	7	7 (Zehnerblock)
3E	8	8 (Zehnerblock)
3F	9	9 (Zehnerblock)

***Tabelle A.2:** Tastaturcodes (Fortsetzung nächste Seite)*

RAW:- Tasten- Code	**Wert ohne Shift-Taste**	**Wert mit Shift-Taste**
40	Leertaste	
41	Backspace	
42	Tabulator	
43	Enter	
44	Return	
45	Escape (Esc)	
46	Del	
47	nicht definiert	
48	nicht definiert	
49	nicht definiert	
4A	–	- (Zehnerblock)
4B	nicht definiert	
4C	Cursor nach oben	Scroll nach unten
4D	Cursor nach unten	Scroll nach oben
4E	Cursor vorwärts	Scroll nach links
4F	Cursor rückwärts	Scroll nach rechts
50	F1	<CSI>10~
51	F2	<CSI>11~
52	F3	<CSI>12~
53	F4	<CSI>13~
54	F5	<CSI>14~
55	F6	<CSI>15~
56	F7	<CSI>16~
57	F8	<CSI>17~
58	F9	<CSI>18~
59	F10	<CSI>19~
5A	nicht definiert	
5B	nicht definiert	
5C	nicht definiert	
5D	nicht definiert	
5E	nicht definiert	
5F	Help	
60	Umstell-Taste (links vom Leerzeichen)	
61	Umstell-Taste (rechts vom Leerzeichen)	
62	Umstell-Taste-Feststeller	

Tabelle A.2: *Tastaturcodes (Fortsetzung nächste Seite)*

RAW:-Tasten-Code	**Wert ohne Shift-Taste**	
63	Control	
64	linke Alt-Taste	
65	rechte Alt-Taste	
66	»AMIGA« (links vom Leerzeichen)	
67	»AMIGA« (rechts vom Leerzeichen)	
68	linke Maustaste	Diese Eingaben
	<nicht übernommen>	werden nur
69	rechte Maustaste	übernommen, wenn
	<nicht übernommen>	die Maus durch
6A	mittlere Maustaste	Intuition in
	<nicht übernommen>	Gameport 1 behandelt wird.
6B	nicht definiert	
6C	nicht definiert	
6D	nicht definiert	
6E	nicht definiert	
6F	nicht definiert	
70-7F	nicht definiert	
80-F8	obenstehende Tasten nicht gedrückt oder bereits losgelassen.	
80 für 00, F8 für 7F.		
F9	letzter Key-Code war ungültig.	
FA	Tastaturpuffer voll	
FB	nicht definiert; reserviert für »Ausfall des Tastaturprozessors«	
FC	Selbsttest der Tastatur fehlgeschlagen.	
FD	Power-up Key Stream Start. Wurden während des Einschaltvorgangs Tasten gedrückt, werden diese zwischen FD und FE gesendet.	
FE	Power-up Key Stream Ende.	
FF	nicht definiert	
FF	Mausaktion, nur im Moment der Aktion. <nicht übernommen>	

***Tabelle A.2:** Tastaturcodes (Schluß)*

Hinweise zur obenstehenden Tabelle:

1. *nicht definiert* bedeutet, daß diese Zahl bei normaler Tastaturbelegung nicht erzeugt wird. Ändern Sie jedoch mit SETMAP diese Belegung, kann dieser Wert jedoch erzeugt werden.

2. <nicht übernommen> bezieht sich auf die Abfrage der Maus-Tasten. Mit der Sequenz <CSI>2{ weisen Sie den Console-Driver an, diese Mausklicks zu melden. Geben Sie dies nicht gesondert an, werden Mausklicks über RAW: nicht gemeldet.

AMIGA

DOS-Handbuch

3. Buch

Das technische Handbuch

Kapitel 9: Das AmigaDOS-File-System

Dieses Kapitel beschreibt das Aufzeichnungsformat, das AmigaDOS beim Beschreiben von Disketten verwendet.

9.1 Die AmigaDOS-File-Struktur

Der AmigaDOS-File-Handler behandelt Disketten, die mit einer einheitlichen Blocklänge formatiert sind. Directories sind mit unbegrenzter Verschachtelungstiefe möglich, dabei kann jedes Directory nur Files, nur Directories oder beides enthalten. Als Verschachtelungs-Art steht nur die Baum-Struktur zur Verfügung – Schleifen sind nicht erlaubt.

Innerhalb dieser Struktur können Sie nahezu alle hardwaremäßig erzeugten Fehler auf der Diskette beseitigen. Die Arbeit damit erleichtert Ihnen ein Disk-Monitor (zum Beispiel DISKED). Daneben benötigen Sie jedoch genaues Wissen um die Aufzeichnungs-Struktur von AmigaDOS. Die nachfolgenden Abschnitte beschreiben die möglichen Block-Typen einer Diskette.

9.1.1 Der Hauptblock einer Diskette

Die Wurzel des Dateien-Baumes ist der ROOT-Block, der erste erstellte Block auf der Diskette. Dieser ROOT-Block ist nahezu identisch zu allen anderen Directories, hat jedoch kein Ursprungs-Directory, und auch der sekundäre File-Typ unterscheidet ihn von allen anderen Blöcken. Im Namen-Feld dieses ROOT-Blockes wird der Name der Diskette eingetragen.

Jeder Block einer Diskette enthält eine Check-Summe, die Summe aller Worte in diesem Block ist Null.

Die folgende Abbildung zeigt den Aufbau des ROOT-Blockes.

0	T.Kurz	Typ
1	0	Header key (immer Null)
2	0	Höchste Sequenz-Nummer (immer Null)
3	HT Größe	Hash-Tabellen-Größe (= Blockgröße-56)
4	0	
5	CECKSUM	Checksumme
6	Hash-	
	Tabelle	
	...	
Länge - 51		
Länge - 50	BMFLAG	Wahr, wenn Bitmap der Disk zulässig.
Länge - 49	Bitmap-	Zeigt auf die Blöcke, die die Bitmap enthalten
	Seiten	
Länge - 24		
Länge - 23	DAYS	Datum und Zeit der letzten Veränderung der Diskette
Länge - 22	MINS	
Länge - 21	TICKS	
Länge - 20	DISK	Name der Diskette als BCPL-String
	NAME	enthält <=30 Zeichen
Länge - 7	CREATEDAYS	Datum und Zeit der Disketten-Formatierung
Länge - 6	CREATEMINS	
Länge - 5	CREATETICKS	
Länge - 4	0	Nächster Eintrag in dieser Hash-chain (immer Null)
Länge - 3	0	Ursprungsdirectory (immer Null)
Länge - 2	0	Erweiterung (immer Null)
Länge - 1	ST.ROOT	Sekundär-Typ zeigt, daß es sich um den ROOT-Block handelt

***Bild 9.1:** Der ROOT-Block einer Diskette*

Der Block besteht also aus 6 Header-Worten, der sogenannten Hash-Tabelle, die bei Wort 6 beginnt, und 50 Worten am Ende des Blocks. In dem Bild sind diese letzten 50 Worte von hinten startend numeriert.

9.1.2 User-Directory-Blocks

Bild 9.2 zeigt den Aufbau eines Blocks, der ein vom Anwender erstelltes Directory enthält.

0	T.SHORT	Typ
1	OWN KEY	Header Key (Pointer zeigt auf sich selbst)
2	0	Höchste Sequenz-Nummer (immer Null)
3	0	
4	0	
5	CHECKSUM	Checksumme
6	Hash-	
	Tabelle	
Länge - 51		
Länge - 50	überzählig	
Länge - 48	PROTECT	Enthält die Protect-Bits
Länge - 47	0	Nicht benutzt (immer Null)
Länge - 46	COMMENT	Der Kommentar als BCPL-String
	..	
Länge - 24		
Länge - 23	DAYS	Datum und Zeit der Erstellung des Directory
Länge - 22	MINS	
Länge - 21	TICKS	
Länge - 20	DIRECTORY	Als BCPL-String gespeichert
	NAME	<=30 Zeichen
	...	
Länge - 4	HASHCHAIN	Nächster Eintrag mit gleichem Hash-Wert
Länge - 3	PARENT	Pointer zurück zum Ursprungs-Directory
Länge - 2	0	Erweiterung (immer Null)
Länge - 1	ST.USERDIR	Sekundär-Typ zeigt, daß es sich um Anwender-Directory handelt.

Bild 9.2: *Blockbelegung eines User-Directory-Blocks*

User-Directory-Blocks tragen den Typ T.SHORT und als Sekundär-Typ die Bezeichnung ST.USERDIRECTORY. Die ersten sechs Worte enthalten die Blocknummer und die Größe der Hash-Tabelle. Die 50 Worte am Ende eines Blocks enthalten das Datum und die Zeit der Erstellung des Directory, einen Zeiger auf das nächste File oder Directory und einen Zeiger auf das Ursprungs-Directory.

Um ein File oder Directory zu finden, wenden Sie zunächst auf dessen Namen eine Hash-Funktion an. Diese Funktion liefert einen Offset in die Hash-Tabelle. Dort findet sich ein Zeiger auf den ersten Block einer Kette von Blöcken, deren Name denselben Hash-Wert ergibt. AmigaDOS liest dann den Block ein, vergleicht dessen Namen mit dem gesuchten Namen und geht zum nächsten Block der Kette, bis der gesuchte Name gefunden ist.

9.1.3 Der Header eines Files

Bild 9.3 zeigt die Belegung des *File-Header-Blocks.*

0	T.SHORT	Typ
1	OWN KEY	Header Key (zeigt auf sich selbst)
2	HIGHEST SEQ	Anzahl von Datenblöcken für das File
3	DATA SIZE	Anzahl der benutzten Datenblöcke
4	FIRST DATA	Erster Datenblock
5	CHECKSUM	Checksumme
6		
	...	
	DATA BLK 3	Liste der Datenblock-Zeiger
	DATA BLK 2	
Länge - 51	DATA BLK1	
Länge - 50	überzählig	
Länge - 48	PROTECT	Protect-Bits
Länge - 47	BYTE SIZE	Gesamtanzahl der Bytes in dem File
Länge - 46	COMMENT	Kommentar als BCPL-String
	...	
Länge - 24		
Länge - 23	DAYS	Datum und Zeit der Erstellung
Länge - 22	MINS	
Länge - 21	TICKS	
Länge - 20	FILE	Name des Files als BCPL-String
	NAME	<=30 Zeichen
	...	
Länge - 4	HASHCHAIN	Nächster Eintrag mit gleichem Hash-Wert
Länge - 3	PARENT	Zeiger zum Ursprungs-Directory
Länge - 2	EXTENSION	Null oder Zeiger zum nächsten Block
Länge - 1	ST.FILE	Sekundärer Filetyp

Bild 9.3: *Die Belegung des File-Header-Blockes.*

Jedes Datenfile beginnt mit einem Header-Block des Typs T.SHORT und des Sekundär-Typs ST.FILE. Anfang und Ende eines Blocks enthalten den Namen, die Zeit der Erstellung und alle Informationen wie ein Directory-Block. Der Hauptteil des Files besteht aus den Datenblöcken, die von eins an aufwärts numeriert sind. AmigaDOS speichert die Adressen dieser Blocks in aufeinanderfolgenden Worten ab Wort 51 (von hinten) im File-Header-

Block. Normalerweise benutzt AmigaDOS nicht alle Speicherzellen für diese Liste, und der letzte Datenblock wird auch nicht voll genutzt.

9.1.4 Der File-List-Block

Enthält ein File mehr Blöcke, als die Blockliste im File-Header-Block aufnehmen kann, wird in das EXTENTION-Feld ein Zeiger auf die nächste von diesem File verwandte Blockliste geschrieben. Bild 9.4 zeigt die Struktur eines solchen File-List-Blockes.

0	T.LIST	Typ
1	OWN KEY	Header Key
2	BLOCK COUNT	Anzahl der Datenblocks in der Block-List
3	DATA SIZE	Wie oben
4	FIRST DATA	Erster Datenblock
5	CHECKSUM	Checksumme
6		
	BLOCK N+3	
	BLOCK N+2	Erweiterte Liste der Datenblock-Zeiger
	...	
Länge - 51	BLOCK N+1	
Länge - 50		
	INFO	(unbenutzt)
	...	
Länge - 4	0	Nächster Eintrag in Hash-Liste
Länge - 3	PARENT	File-Header-Block dieses Files
Länge - 2	EXTENSION	Nächster Extension-Block
Länge - 1	ST.FILE	Sekundärer Filetyp

Bild 9.4: *Der File-List-Block*

Der Aufbau eines File-List-Blocks ist nahezu identisch mit dem des File-Header-Blocks. File-List-Blocks gibt es zu jeder Datei so viele, wie nötig sind, um alle Blockadressen der Datenblöcke dieses Files aufzunehmen. Sie werden über das Feld EXTENSION miteinander verkettet.

9.1.5 Der Datenblock

Das folgende Bild zeigt die Struktur eines Datenblocks:

0	T.DATA	Typ
1	HEADER	Header Key
21	SEQ SUM	Anordnungszahl
3	DATA SIZE	Datengröße
4	NEXT DATA	Nächster Datenblock
5	CHECKSUM	Checksumme
6		
	DATA	Daten

Bild 9.5: Der Datenblock

Datenblöcke enthalten lediglich sechs Informationen zur Dateiverwaltung, jede in einem Wort gespeichert. Diese sechs Worte enthalten:

- den Typ des Files
- einen Zeiger zum File-Header-Block
- die Anordnungszahl des Blocks innerhalb eines Files
- die Anzahl an Worten
- einen Zeiger zum nächsten Datenblock
- die Checksumme

Die restlichen Worte des Blocks enthalten die eigentlichen Daten, die in dem jeweiligen File abgespeichert sind. Alle Datenblöcke eines Files werden mit diesen Daten gefüllt (Anzahl der Worte: Blockgröße-6). Lediglich der letzte kann freien Speicherplatz enthalten. Dessen NEXT DATA-Block enthält dann Null als Zeichen dafür, daß kein weiterer Block existiert.

Die Angabe Wort in dem vorangegangenen Abschnitt bezieht sich natürlich nicht auf reale Worte, sondern auf die Datenlänge Wort (16 Bit)!

Kapitel 10: Die Struktur von Binär-Files

10.1 Einführung

Kapitel 10 beschreibt den genauen Aufbau eines binären Objektfiles, wie es von einem Assembler oder Compiler erzeugt wird. Ebenso beschrieben wird das Format eines binären Load-Files, das vom Linker erstellt und vom Ladeprogramm in den Hauptspeicher gelesen wird. Das Format der Load-Files unterstützt das Overlay von Programmen. Getrennt von der Beschreibung der Load-Files erklärt dieses Kapitel den Einsatz gemeinsamer Symbole, absoluter externer Referenzen und von Programm-Einheiten.

10.1.1 Terminologie

Einige der in den folgenden Abschnitten verwendeten Fachbegriffe werden nun kurz erklärt.

Externe Referenzen
Sie können Namen verwenden, um Referenzen zwischen einzelnen Programmeinheiten zu kennzeichnen. Unter ***externen Referenzen*** verstehen wir dabei Namen, die in einem Programmteil definiert werden, die aber aus anderen Teilen ebenfalls verwendet werden können. Da jeder einzelne Programmteil separat assembliert werden kann, können diesen externen Labels natürlich noch keine Adressen zugeordnet werden. Dies geschieht erst durch den Linker, der die einzelnen Programmteile (die durch den Assembler erzeugten Objekt-Files) zusammenlinkt.

Namen, die als externe Referenzen benutzt werden, können von der Datenstruktur her 16 Mbyte lang sein, werden jedoch vom Linker auf 255 Zeichen gekürzt. Linken Sie Objektfiles zu einem einzelnen ladbaren File, muß jeder einzelnen externen Referenz eine externe Symboldefinition zugeordnet sein. Diese externe Referenz kann vom Datentyp Byte,

Wort oder Langwort sein. Externe Definitionen beziehen sich auf relozierbare (verschiebbare) Werte, absolute Werte oder residente Bibliotheken. Relozierbare Werte beziehen sich auf den PC-relativen Adreßmodus und werden vom Linker verwaltet. Wann immer ein Programm relozierbare Referenzen mit der Datenlänge Langwort enthält, wird die Relokation erst beim Laden des Programmes durch einen speziellen Relokator des Amiga-Betriebssystems durchgeführt.

Objekt-File
Ein Assembler oder Compiler erzeugt ein binäres Ausgabe-File, Objekt-File genannt. Ein Objekt-File besteht aus einer oder mehreren Programmeinheiten und kann externe Referenzen zu anderen Modulen enthalten.

Ladbares File (Load-File)
Der Linker erzeugt ein binäres Ausgabe-File aus einer beliebigen Anzahl von Objekt-Files. Dieses Ausgabe-File wird auch *ladbares File* genannt. Ein ladbares File enthält keine unaufgelösten externen Referenzen mehr.

Programmeinheit
Eine Programmeinheit ist das kleinste vom Linker bearbeitbare Element. Eine Programmeinheit kann ein oder mehrere Module (*Hunks*) enthalten; ein Objekt-File wiederum enthält eine oder mehrere Programmeinheiten. Findet der Linker eine externe Referenz, während er die Bibliotheks-Funktionen durchforstet, wird die gesamte Programmeinheit, in der die Referenz definiert wird, in das ladbare File eingebunden. Ein Assembler erzeugt ein einziges Programm aus einem Quell-File (das ein oder mehrere Module enthalten kann). Ein Compiler erzeugt meist eine Programmeinheit für jedes Unterprogramm, Hauptprogramm oder jeden Datenblock. Die Numerierung der Module beginnt in jeder Programmeinheit mit Null, auf andere Programmeinheiten kann nur mit externen Referenzen verwiesen werden.

Module (hHnks)
Ein Modul besteht aus Daten- oder Programmblöcken, Relokations-Informationen und eventuell einer Liste von definierten oder benötigten externen Symbolen. Daten-Module enthalten initialisierte oder nicht initialisierte Daten (BSS). BSS-Module können externe Definitionen enthalten. Sie können jedoch keine Referenzen auf externe Symbole oder relozierbare Werte enthalten. Initialisierte Datenblöcke innerhalb eines Overlay-Files sollten nicht geändert werden (wie Konstanten behandelt werden), da sie während des Overlay-Prozesses immer wieder neu von der Diskette gelesen werden können. Eventuelle Änderungen gehen dabei verloren.

Module können benannt werden oder unbenannt bleiben, und sie können eine Symbol-Tabelle für Debugging-Zwecke enthalten. Auch Informationen zum Debugging mit höheren Programmiersprachen können vom Linker eingebunden werden. Die Module innerhalb einer Programmeinheit bekommen alle eine Nummer. Das erste Modul trägt die Nummer Null.

Residente Libraries (immer verfügbare Bibliotheken)
Ladbare Files werden oft auch *Bibliotheken* oder *Libraries* genannt. Solche Bibliotheken können entweder immer im Speicher bereitgehalten (*residente Libraries*) oder als Teil der Exec-Funktion OpenLibrary() bei Bedarf geladen werden. Auf residente Libraries können Sie in Ihrem Programm durch externe Referenzen verweisen. Diese Verweise werden in ein Modul geschrieben, das nur eine Liste der residenten Libraries ohne weiteren Code enthält. Dieses Modul erstellen Sie, indem Sie ein File assemblieren, das nichts außer absoluten externen Symbolen enthält, und dieses mit einem speziellen Programm bearbeiten, das die absoluten Definitionen in Definitionen für residente Libraries umwandelt. Der Linker verwendet den Modul-Namen dann als Namen der residenten Library und bindet entsprechende Aufrufe in den Programmcode ein, so daß bei der Programmausführung die residente Library automatisch geöffnet wird, bevor sie das erste Mal verwendet wird.

Scanned Libraries
Eine *Scanned Library* besteht aus Objekt-Files, die Programmeinheiten enthalten, die nur dann geladen werden, wenn durch eine externe Referenz auf sie verwiesen wird. Scanned Libraries sind also nicht resident, sondern werden erst dann geladen, wenn sie benötigt werden. Objekt-Files können als Libraries definiert und dem Linker als primäres Input-File übergeben werden. In diesem Fall werden alle Programmeinheiten, die die Bibliothek enthält, beim Linken mitverarbeitet. Beachten Sie, daß Objekt-Files verkettet werden können.

Knoten
Ein Knoten besteht aus mindestens einem Modul. Ein Overlay-Load-File enthält einen Wurzelknoten, der im Speicher gehalten wird, solange das Programm läuft, sowie mehrere Overlay-Knoten, die bei Bedarf in den Speicher gelesen werden.

10.2 Die Objekt-File-Struktur

Als *Objekt-File* wird die Ausgabe eines Assemblers oder eines Compilers bezeichnet. Um das Objekt-File zum ausführbaren Programm zu machen, werden zunächst alle externen Referenzen aufgelöst. Diese Arbeit übernimmt der Linker. Ein Objekt-File besteht aus einer oder mehreren Programmeinheiten. Jede Programmeinheit beginnt mit einem Header. Dann folgt eine Anzahl Module, die wiederum aus Blöcken unterschiedlichen Typs bestehen. Jeder Block beginnt mit einem Langwort, das den Typ des Blocks enthält. Dann folgt eine Null oder weitere Langworte. Jeder Block wird auf volle Langwortlänge erweitert. Der Header einer Programmeinheit beginnt ebenfalls mit einem Block dieses Formates.

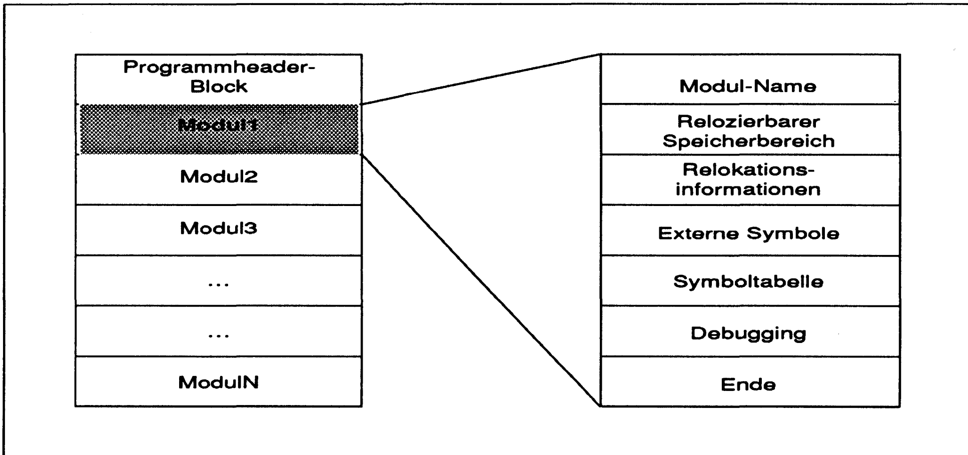


Bild 10.1: *Objekt-File-Struktur*

Eine Programmeinheit enthält also:

- Den Header-Block der Programmeinheit
- Eine Reihe von Modulen

Die Grundstruktur eines Moduls besteht aus:

- Einem Speicherblock für den Modul-Namen
- Einem relozierbaren Block
- Einem Block mit Informationen zur Relokation
- Einem Block mit den Informationen zu externen Symbolen
- Einem Block mit der Symboltabelle
- Einem Block mit Debugging-Informationen
- Dem Endblock

Alle Blöcke mit Ausnahme des Endblocks können allerdings auch fortfallen.

Die folgenden Abschnitte beschreiben das Format jedes dieser Blöcke. Die Werte, die als Angabe des Blocktyps übergeben werden, sind in dezimaler und hexadezimaler Form nach dem Modul-Namen abgedruckt. Zum Beispiel wird dem »hunk_unit« (*hunk* bedeutet *Modul*) als Definition der Wert 999 dezimal oder 3E7 hexadezimal zugewiesen.

10.2.1 hunk_unit (999/3E7)

Dieses Modul zeigt den Beginn einer Programmeinheit an. Das erste Wort enthält den Typ des Moduls, gefolgt von der Länge des Namens der Programmeinheit in Langworten, wiederum gefolgt vom Namen der Programmeinheit in Langworten, bei Bedarf bis zur

nächsten Langwortgrenze aufgefüllt mit Nullen. Grafisch dargestellt sieht hunk_unit wie folgt aus:

hunk_unit
N
N Langworte mit dem Namen ...

***Bild 10.2:** hunk_unit (999/3E7)*

10.2.2 hunk_name (1000/3E8)

Dieser Block definiert den Namen des Moduls. Namen sind optional. Findet der Linker mehrere Module mit gleichem Namen, werden sie zu einem Modul verknüpft. Beachten Sie bitte, daß 8- oder 16-Bit-externe-Referenzen nur unter Modulen mit gleichem Namen aufgelöst werden können. Jede externe Referenz in einem ladbaren File, die auf ein anderes Module verweist, muß als 32-Bit-Referenz übergeben werden. Die Module eines ladbaren Files werden nicht hintereinander, sondern eventuell verteilt in den Speicher geladen. Sie können deshalb auch nicht davon ausgehen, daß sie untereinander nur 32 Kbyte Abstand haben. Bedenken Sie bitte auch, daß alle Blöcke auf eine ganze Zahl von Langworten aufgefüllt werden. Grafisch dargestellt sieht hunk_name so aus:

hunk_name
N
N Langworte mit dem Namen ...

***Bild 10.3:** hunk_name (1000/3E8)*

10.2.3 hunk_code (1001/3E9)

Ein solcher Hunk definiert einen Block, der Code enthält und in den Speicher geladen und eventuell reloziert werden kann. Sein Format sieht wie folgt aus:

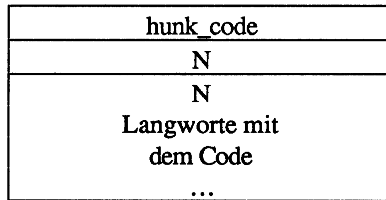


Bild 10.4: *hunk_code (1001/3E9)*

10.2.4 hunk_data (1002/3EA)

Definiert einen initialisierten Datenblock, der in den Speicher geladen und dort reloziert werden kann. Der Linker verändert diese Blöcke nicht, wenn sie Teil eines Overlay-Knotens sind. Während des Overlay-Vorgangs werden sie also erneut in den Speicher geladen. Grafisch dargestellt sieht hunk_data wie folgt aus:

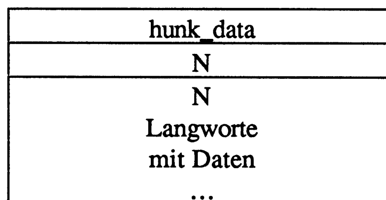


Bild 10.5: *hunk_data (1002/3EA)*

10.2.5 hunk_bss (1003/3EB)

Mit diesem Hunk wird ein nicht initialisierter Block im Arbeitsspeicher definiert, der später nachgeladen wird. Diese Blockform wird zum Beispiel für Stacks oder für FORTRAN-COMMON-Blöcke verwendet. Innerhalb dieser Blöcke kann nicht reloziert werden. Symboldefinitionen sind aber zulässig. Grafisch dargestellt sieht hunk_bss wie folgt aus:

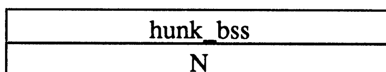


Bild 10.6: *hunk_bss (1003/3EB)*

Dabei ist N die Blocklänge in Langworten. Beim Laden wird der Speicher von BSS-Blöcken mit Null beschrieben. Der relozierbare Block innerhalb eines Moduls muß vom Typ hunk_code, hunk_data oder hunk_bss sein.

10.2.6 hunk_reloc32 (1004/3EC)

Ein hunk_reloc32-Block liefert Informationen zur 32-Bit-Relokation, die der Linker mit dem jeweils aktuellen relozierbaren Block durchführt. Die Informationen zur Relokation beziehen sich auf eine Stelle im aktuellen Modul oder auf eine innerhalb derselben Programmeinheit. Alle Module innerhalb einer Programmeinheit sind – beginnend mit Null – durchnumeriert. Der Linker addiert die Basis-Adresse des angegebenen Moduls zu den Langworten im vorhergehenden relozierbaren Block, der eine Liste mit Offsets enthält. Diese Offset-Liste enthält nur bestimmte Module und und als Abschluß eine Null. Grafisch dargestellt sieht hunk_reloc32 wie folgt aus:

hunk_reloc32
N1
Modul Nummer 1
N1 Offsets ...
N2
Modul Nummer 2
N2 Offsets ...
Nn
Modul Nummer n
Nn Offsets ...
0

***Bild 10.7:** hunk_reloc32 (1004/3EC)*

10.2.7 hunk_reloc16 (1005/3ED)

Ein hunk_reloc16-Block liefert Informationen zur 16-Bit-Relokation, die der Linker mit dem jeweils aktuellen relozierbaren Block durchführt. Diese Information bezieht sich auf 16-Bit-PC-relative Referenzen auf andere Module innerhalb der Programmeinheit. Referenzen dürfen sich nur auf gleichnamige Module beziehen, da der Linker nur diese zu einer Einheit verkettet, die zusammen geladen werden. Das Format von hunk_reloc16 entspricht dem von hunk_reloc32.

10.2.8 hunk_reloc8 (1006/3EE)

Ein hunk_reloc8-Block liefert Informationen zur 8-Bit-Relokation, die der Linker mit dem jeweils aktuellen relozierbaren Block durchführt. Diese Information bezieht sich auf 8-Bit-PC-relative Referenzen auf andere Module innerhalb der Programmeinheit. Referenzen dürfen sich nur auf gleichnamige Module beziehen, da der Linker nur diese zu einer Einheit verkettet, die zusammen geladen werden. Das Format von hunk_reloc8 entspricht dem von hunk_reloc32.

10.2.9 hunk_ext (1007/3EF)

Dieser Block enthält Informationen zu externen Symbolen. Eingetragen werden sowohl die definierten Symbole als auch die externen Symbole, die dieses Modul aus anderen benötigt. Hier das Format:

hunk_ext
Symbol Daten-einheit
Symbol Daten-einheit
...
0

Bild 10.8: *hunk_ext (1007/3EF)*

Dabei wird eine *Symbol-Dateneinheit* für jedes Symbol verwandt. Der Block endet mit einem Langwort mit dem Wert Null.

Jede Symbol-Dateneinheit enthält ein Byte zur Typdefinition, die Länge des Symbol-Namens in drei Bytes, den Symbol-Namen selbst und weitere Daten. Die Länge des Symbol-Namens geben Sie in Langworten an, der Symbol-Name selbst wird mit Nullen auf das nächste Langwort aufgefüllt.

Das Typ-Byte definiert das Symbol als Definition oder Referenz. AmigaDOS verwendet für Symbol-Definitionen die Werte 0–127, die Werte 128–255 für Referenzen.

Folgende Werte sind definiert:

Name	**Wert**	**Bedeutung**
ext_symb	0	Symbol-Tabelle – siehe Symbol-Block weiter unten
ext_def	1	Relozierbare Definition
ext_abs	2	Absolute Definition
ext_res	3	Residente Library-Definition
ext_ref32	129	32-Bit-Referenz zu einem Symbol
ext_common	130	32-Bit-Referenz zu einem COMMON-Block
ext_ref16	131	16-Bit-Referenz zu einem Symbol
ext_ref8	132	8-Bit-Referenz zu einem Symbol

Tabelle 10.1: *Externe Symbole*

Der Linker gibt bei anderen Werten eine Fehlermeldung aus. Bei ext_def wird nur ein Datenwort verwandt. Es enthält nur den Offset des Symbols relativ zum Beginn des Moduls. Ext_abs enthält ebenfalls nur ein Datenwort, den absoluten Wert, der zum Code hinzugefügt wird. Der Linker behandelt den Wert von ext_res wie den von ext_def, geht aber davon aus, daß der Modul-Name ein Library-Name ist, und kopiert den Namen in das ladbare File. Den Typ-Bytes ext_ref32, ext_ref16 und ext_ref8 folgen jeweils ein Wert und eine Liste von Referenzen, die als Offset vom Beginn des Moduls angegeben sind.

Der Typ ext_common hat die gleiche Struktur wie ein COMMON-Block. Der Linker verarbeitet als COMMON definierte Symbole wie folgt: Findet er eine Definition für ein Symbol, das eine Referenz zu COMMON hat, verwendet er diesen Wert. Andernfalls belegt er den für jede COMMON-Symbol-Referenz als maximal angegebenen bss-Speicherplatz.

Der Linker bearbeitet externe Referenzen abhängig vom Typ der dazugehörigen Definition. Absolute Werte werden zu Langworten oder Byte-Feldern addiert. Paßt dieser (vorzeichenbehaftete) Wert nicht zur Definition, wird ein Fehler gemeldet. Bei relozierbaren 32-Bit-Referenzen wird der Wert des zugehörigen Symbols addiert und ein Relocation-Record für den Lader erstellt. 16- und 8-Bit-Referenzen werden als PC-relativ behandelt und sind nur für Relokationen innerhalb gleichnamiger Module zulässig. Solche PC-relative Referenzen werden nicht aufgelöst, wenn die Referenz und die Definition zu weit voneinander entfernt sind. Der Linker kann auf residente Libraries nur zugreifen, wenn 32-Bit-Referenzen dafür definiert wurden.

Die Symbol-Dateneinheiten sehen wie folgt aus:

ext_def/abs/res

Typ I Name Länge NL
NL Langworte des Symbolnamens ...
Symbolwert

ext_ref32/16/8

Typ I Name Länge NL
NL Langworte des Symbolnamens ...
Anzahl der Referenzen NR
NL Langworte der Symbolreferenzen ...

ext_common

130 I Name Länge NL
NL Langworte des Symbolnamens ...
Größe des COMMON-Blocks
Anzahl der Referenzen NR
NR Langworte der Symbolreferenzen ...

Bild 10.9: *Symbol Dateneinheit*

10.2.10 hunk_symbol (1008/3E0)

Damit Sie einen symbolischen Debugger verwenden können, können Sie mit diesem Typ von Block eine Symboltabelle an ein Modul anhängen. Werden mehrere Module gleichen Namens zusammengelinkt, verknüpft der Linker auch deren Symboltabellen. Symboltabellen werden nicht in den Speicher geladen. Bei Bedarf muß sie der Debugger von der Diskette lesen. Das Format des Symboltabellen-Blocks entspricht dem des externen Symbol-Informations-Blocks mit Symboltabellen-Einheiten für jeden verwendeten Namen. In den Symbol-Dateneinheiten wird der Typencode Null verwendet. Der Wert des Symbols ist der

Offset des Symbols vom Anfang des Moduls. Grafisch dargestellt sieht hunk_symbol wie folgt aus:

hunk_symbol
Symbol Daten- einheit ...
0

Bild 10.10: *hunk_symbol (1008/3F0)*

dabei hat jede Symbol-Dateneinheit folgendes Format:

0 | Name Länge NL
NL Langworte des Symbolnamens ...
Symbolwert

Bild 10.11: *Symbol Dateneinheit*

10.2.11 hunk_debug (1009/3F1)

AmigaDOS stellt diesen Debug-Block zur Verfügung, um weitere Informationen für Debugging-Zwecke in einem ladbaren File ablegen zu können. Compiler können zum Beispiel eine Beschreibung der Datenstrukturen zur Verfügung stellen, die von Hochsprachen-Debuggern benötigt werden. Diese Informationen können im Debug-Block enthalten sein. AmigaDOS schreibt für diesen Block nur den Anfang vor. Er beginnt mit der Typ-Kennung hunk_debug als Langwort, im zweiten Langwort steht die Länge des Blocks. Das Format der restlichen Daten ist nicht definiert.

Grafisch dargestellt sieht hunk_debug wie folgt aus:

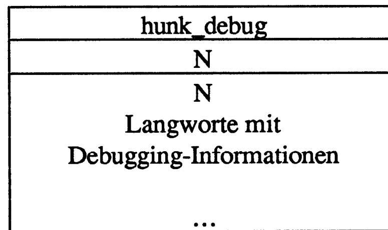


Bild 10.12: *hunk_debug (1009/3F1)*

10.2.12 hunk_end (1010/3F2)

Dieser Block definiert das Ende des Moduls. Er enthält nur das Langwort hunk_end.

10.3 Ladbare Files (Load-Files)

Das Format eines Load-Files (des Ausgabe-Files des Linkers) ist ähnlich dem eines Objekt-Files. Es enhält eine Anzahl von Modulen mit gleichem Format wie die Module in Objekt-Files. Der hauptsächliche Unterschied ist, daß ein Load-File niemals einen Block mit externen Symbol-Informationen enthält. Alle Symbolreferenzen sind aufgelöst, die Programmeinheit-Informationen werden nicht übernommen. Ein einfaches Load-File (nicht im Overlay erstellt) enthält einen Header-Block, der die Anzahl der Module innerhalb des Load-Files und alle residenten Libraries angibt. Diesem Block folgen die Module, die jedes eventuell aus mehreren Eingabe-Modulen mit gleichem Namen bestehen. Diese ganze Struktur wird als ein Knoten behandelt. Zusätzlich kann im Load-File auch die Overlay-Information enthalten sein. In diesem Fall folgt dem primären Knoten die Overlay-Tabelle. Ein spezieller Break-Block trennt die Overlay-Knoten voneinander. Diese Overlay-Struktur kann wie folgt dargestellt werden, wobei optionale Teile mit einem Stern (*) gekennzeichnet sind.

- Primärer Knoten
- Overlay- Tabelle (*)
- Overlay-Knoten, getrennt mit Break-Blöcken (*)

Die Relokations-Blöcke innerhalb der Module sind immer vom Typ hunk_reloc32. Die Relokation geschieht während des Ladens. Dabei werden nicht nur die Relokationen mit hunk_reloc32, sondern auch die durch die Auflösung der externen Symboltabelle durchgeführt.

Jede externe Referenz im Objekt-File wird wie folgt behandelt: Der Linker durchsucht den primären Input nach passenden externen Definitionen. Findet er diese nicht, durchforstet er die Scanned Libraries und bindet die gesamte Programmeinheit, in dem die Definition steht, in das Load-File ein. Dadurch können wieder externe Referenzen auftreten. Am Ende des ersten Durchlaufes kennt der Linker alle externen Definitionen und alle Module, die dem Programm angehören sollen. Im zweiten Durchlauf werden die Langworte der externen Referenzen so verändert, daß sie den benötigten Offset zu dem Modul herstellen, in dem das Symbol definiert ist. Wenn alle Module geladen sind, wird im Relokations-Block ein Eintrag erzeugt, der die Basisadresse jedes Moduls enthält, das das Symbol definiert. Dieses Verfahren gilt auch für residente Libraries.

Bevor der Lader diese Modul-übergreifenden Referenzen auflösen kann, muß er die Anzahl und Größe aller Module des jeweiligen Knotens kennen. Der weiter unten beschriebene Header-Block enthält diese Informationen. Das Load-File enthält auch eventuell Overlay-Informationen in einem speziellen Block, der Overlay-Tabelle. Break-Blöcke trennen die einzelnen Overlay-Knoten.

10.3.1 hunk_header (1011/3F3)

Dieser Block enthält die Informationen über die Anzahl und Länge aller zu ladenden Module. Ebenso sind die Namen aller residenten Libraries, die für diesen Knoten geöffnet werden müssen, enthalten. Das Format von hunk_header ist in Bild 10.13 dargestellt. Der erste Teil des Blocks enthält die Namen aller zu öffnenden residenten Libraries, die in diesen Knoten eingebunden werden sollen. Jeder Name besteht aus einem Langwort mit der Länge des Namens der residenten Library und dem Namen selbst in Langworten, aufgefüllt bis zur nächsten Langwortgrenze mit dem Wert Null. Die Liste der Namen endet mit einem Langwort mit dem Wert Null. Die residenten Libraries werden in der angegebenen Reihenfolge geöffnet.

Wird ein primärer Knoten geladen, erstellt der Lader eine Tabelle im Arbeitsspeicher, in der die Reihenfolge aller geladenen Module gespeichert wird. Die Tabelle muß groß genug sein, um alle Module, auch die in den Overlays, unterzubringen. In diese Tabelle wird auch eine Kopie der Modul-Tabelle zusammen mit allen residenten Libraries geschrieben. Das nächste Langwort im Header enthält die Größe dieser Tabelle (Anzahl aller Module plus 1) in einem Langwort. Das nächste Langwort F verweist auf den ersten Wert in der Modul-Tabelle, die vom Lader benutzt werden soll. Für einen primären Knoten, der keine residenten Libraries eingebunden bekommt, ist dieser Wert Null; andernfalls enthält er die Anzahl von Modulen in den residenten Libraries. Der Lader kopiert diese Einträge der Modul-Tabelle zusammen mit der Library, gefolgt vom Aufruf der Funktion OpenLibrary(). Bei einem Overlay-Knoten ist dieser Wert die Anzahl der Module in allen residenten Libraries und die Anzahl aller bereits in höherstehenden Knoten geladenen Module.

Das nächste Langwort L zeigt auf das letzte zu ladende Modul in diesem Aufruf des Laders. Die Gesamtanzahl der geladenen Module ist dann L-F+1.

hunk_header
N1
N1 Langworts des Namens ...
N2
N2 Langworte des Namens ...
0
Größe der Tabelle
Erstes Modul F
Letztes Modul L
L-F+1 Größenangaben ...

***Bild 10.13:** hunk_header (1010/3F3)*

Der Header-Block enthält dann L-F+1 Langworte. Jedes zeigt die Größe des jeweiligen Moduls an. Damit wird der Lader in die Lage versetzt, den gesamten benötigten Speicherplatz zu belegen, bevor das erste Modul geladen wird. Es steht dann genug Speicher zur Verfügung, um alle Symbole aufzulösen. Ist ein hunk_bss mit einer Größe Null vorhanden, verwendet der Lader eine Systemvariable als Größe dieses Moduls, wie in der Beschreibung von hunk_bss erläutert.

10.3.2 hunk_overlay (1013/3F5)

Der Block mit der Overlay-Tabelle zeigt dem Lader an, daß ein Programm im Overlay-Verfahren erstellt werden soll, und enthält alle dazu nötigen Informationen. Trifft der Lader auf diesen Block, bearbeitet er die Tabelle und kehrt dann zur weiteren Programm-Bearbeitung zurück. Dabei bleiben alle Input-Kanäle geöffnet.

Grafisch dargestellt sieht hunk_overlay wie folgt aus:

hunk_overlay
Größe der Tabelle
M+2
M+1 Nullen ...
Overlay Daten- Tabelle ...

Bild 10.14: *hunk_overlay (1013/3F5)*

Das erste Langwort ist die obere Grenze der gesamten Overlay-Tabelle in Langworten.

M ist die maximale Anzahl von Verzweigungen im Overlay-Baum; die Wurzel hat den Wert Null. Die nächsten M+1 Worte bilden das Gerüst der Overlay-Tabelle.

Der Rest des Blocks ist die Overlay-Datentabelle, eine Reihe von 8-Wort-Einträgen; eines für jedes Overlay-Symbol. Wird als maximale Overlay-Nummer Null eingesetzt, ist die Größe der Overlay-Datentabelle (0+1)*8, da die erste Overlay-Nummer Null ist. Die gesamte Größe der Overlay-Tabelle ist dann (0+1)*8+M+1.

10.3.3 hunk_break (1014/3F6)

Ein Break-Block zeigt das Ende eines Overlay-Knotens an. Er besteht aus einem Langwort mit dem Wert hunk_break.

10.4 Beispiele

Der folgende Abschnitt zeigt anhand kleiner Code-Segmente, wie Linker und Lader externe Symbole bearbeiten:

```
                                IDNT        A
                                XREF        BILLY, JOHN
                                XDEF        MARY
* Das nächste Langwort soll reloziert werden
0000' 0000 0008                 DC.L        FRED
0004' 123C 00FF                 MOVE.B      #$FF,D1
0008' 7001          FRED        MOVEQ       #1,D0
*Externer Einsprung
000A' 4E71          MARY        NOP
000C' 4EB9 0000 0000                        JSR    BILLY     Externer Aufruf
0012' 2239 0000 0000                        MOVE.L           JOHN,D1  Externe Referenz
                                END
```

erzeugt folgendes Objektfile:

```
hunk_unit
00000001            Größe in Langworten
41000000            Name, zum Langwort aufgefüllt mit Nullen
hunk_code
00000006            Größe in Langworten
00000008 123C00FF 70014E71 4EB90000 00002239 00000000
hunk_reloc32
00000001            Nummer in Modul 0
00000000            Modul 0
00000000            Offset zur Relokation
00000000            Null zum Ende
hunk_ext
01000001            XDEF, Größe: ein Langwort
4D415259            MARY
0000000A            Offset der Definition
81000001            XREF, Größe: ein Langwort
4A4F484E            JOHN
00000001            Anzahl der Referenzen
00000014            Offset der Referenzen
81000002            XREF, Größe: zwei Langworte
42494C4C            BILLY
59000000            (Aufgefüllt mit Nullen)
```

```
00000001            Anzahl der Referenzen
0000000E            Offset der Referenzen
00000000            Ende des externen Blocks
hunk_end
```

Das entspricht dem folgenden Programm:

```
                                  IDNT         B
                                  XDEF         BILLY, JOHN
                                  XREF         MARY
0000' 2A3C AAAA AAAA              MOVE.L       #$AAAAAAAA,D5
* Externer Einsprung
0006' 4E71           BILLY        NOP
* Externer Einsprung
0008' 7201           JOHN         MOVEQ        #1,D1
* Aufruf der externen Referenz
000A' 4EF9 0000 0000              JMP          MARY
                                  END
```

Der dazugehörige Output-Code sieht so aus:

```
hunk_unit
00000001            Größe in Langworten
42000000            Name des Teiles
hunk_code
00000004            Größe in Langworten
2A3CAAAA AAAA4E71 72014EF9   00000000
hunk_ext
01000001            XDEF, Größe: ein Langwort
4A4F484E            JOHN
00000008            Offset der Definition
01000002            XDEF, Größe: zwei Langworte
42494C4C            BILLY
59000000            (Aufgefüllt mit Nullen)
00000006            Offset der Definition
81000001            XREF, Größe: ein Langwort
4D415259            MARY
00000001            Anzahl der Referenzen
0000000C            Offset der Referenzen
00000000            Ende des externen Blocks
hunk_end
```

Nachdem der Linker das Ganze bearbeitet hat, erhalten Sie folgendes Load-File:

```
hunk_header
00000000            Kein Modul-Name
00000002            Größe der Modul-Tabelle
00000000            Erstes Modul
00000001            Letztes Modul
00000006            Länge von Modul 0
00000004            Länge von Modul 1
hunk_code
00000006            Länge des Codes in Langworten
00000008 123C00FF 70014E71 4EB90000 00062239 00000008
hunk_reloc32
00000001            Nummer in Modul 0
00000000            Modul 0
00000000            Offset der Relokation
00000002            Nummer in Modul 1
00000001            Modul 1
00000014            Offset der Relokation
0000000E            Offset der Relokation
00000000            Null zum Ende
hunk_end
hunk_code
00000004            Länge des Codes in Langworten
2A3CAAAA AAAA4E71 72014EF9 0000000A
hunk_reloc32
00000001            Nummer in Modul 0
00000000            Modul 0
0000000C            Offset der Relokation
00000000            Null zum Ende
hunk_end
```

Wird dieser Code vom Lader in den Speicher gebracht, liest er den Header-Block ein und erstellt eine Modul-Tabelle von zwei Langworten. Dann wird eine Systemroutine aufgerufen, die zwei Bereiche Speicher von sechs und vier Langworten Größe bereitstellt. Nehmen wir an, diese Bereiche beginnen bei den Speicherstellen 3000 und 7000, so enthält die Modul-Tabelle diese beiden Werte.

Der Lader liest das erste Modul und schreibt den Code nach 3000; dann wird die Relokation durchgeführt. Der erste Begriff bezieht sich auf Modul 0, also wird 3000 zu dem Langwort von Offset 0 addiert. Der Wert wird dann von 00000008 nach 00003008 gebracht. Der zweite Begriff bezieht sich auf Modul 1. Obwohl noch nicht geladen, wissen wir, daß diese Werte nach Speicherplatz 7000 gebracht werden sollen. Deshalb wird es zu den Werten in

den Speicherstellen 300E und 3014 addiert. Beachten Sie, daß der Linker die Offsets 00000006 und 00000008 bereits in die Referenz in Modul 0 eingesetzt hat, deshalb entsteht der korrekte Offset zur Definition in Modul 1. Diese Langworte bestimmen das Ende der externen Referenzen und enthalten nun die Werte 00007006 und 00007008, die korrekt sind, sobald das zweite Modul geladen ist.

Genauso lädt der Lader das zweite Modul nach Speicherstelle 7000. Die Informationen zur Relokation ändern das Langwort bei 700C von 0000000A (dem Offset von MARY im ersten Modul) nach 0000300A (der Adresse von MARY im Speicher – im zweiten Modul).

Der Lader bearbeitet Referenzen auf residente Libraries genauso. Er kopiert lediglich nach dem Öffnen der Library die Adressen der Module, aus denen die Library besteht, an den Anfang der Modul-Tabelle. Danach werden alle Referenzen, die auf die Library verweisen, korrigiert, indem er die Basisadresse des jeweiligen Moduls aufaddiert, um den richtigen Offset zu erhalten.

Kapitel 11: Die AmigaDOS-Datenstrukturen

Dieses Kapitel beschreibt die Datenstrukturen, die der Amiga im Speicher und in Files verwendet. Nicht erklärt wird dagegen das Aufzeichnungsformat auf Diskette. Das geschah bereits in Kapitel 9.

11.1 Einführung

AmigaDOS stellt interne Devices (Geräte) geräteunabhängiger Ein-/Ausgabe zur Verfügung. Dazu wird ein *Handler-Prozeß* für jedes Device erstellt. Diese Prozesse erkennen alle die Ein-/Ausgabe-Anforderungen und wandeln diese in ein jeweils Device-spezifisches Signal um. Alle Ein- und Ausgaben gehen deshalb nicht direkt zum Device, sondern zum *Handler-Prozeß*. Jedoch sind auch direkte Zugriffe zum Device möglich, wenn dies ausnahmsweise nötig sein sollte. Dieses Kapitel beschreibt die Datenstrukturen von AmigaDOS, einschließlich der Formate von Prozessen und den Zugriffen auf den Handler.

Zusätzlich zu den normalen Amiga-Werten wie LONG und APTR benutzt AmigaDOS noch BPTR. BPTR ist ein BCPL-Pointer auf die Adresse eines Langwortes geteilt durch vier. Um in C einen BPTR zu verwenden, verschieben Sie ihn einfach zweimal nach links und erhalten dadurch einen Pointer. Zum Erstellen eines BPTR wird mit Allocmem ein Speicherbereich reserviert oder eine Struktur innerhalb des Stack angesprochen. Dazu müssen Sie aber sicher sein, nur Langworte auf den Stack geschrieben zu haben (der normale Stack ist auf Worte ausgerichtet). Dieser Pointer wird dann zweimal nach rechts verschoben, um BPTR zu erzeugen.

Als Stringfunktion kommt noch BSTR, ein BCPL-String dazu. BSTR besteht aus einem Pointer zur Speicherstelle, die die Länge des Strings im ersten Byte und die Gesamtzahl der Bytes im String im zweiten Byte enthält.

In diesem Kapitel taucht ein *globaler Vektor* auf. Der globale Vektor ist eine von BCPL verwendete Sprungtabelle und ein Pointer zum Standard-Vektor des Amiga. Einige Prozesse verwenden allerdings eigene globale Vektoren, die vom Standard-Vektor verschieden sind.

Die Definitionen für die folgenden Strukturen sind in den Files DOS.H und DOSEXTENS.H für die Sprache C enthalten. Die dazugehörigen Assemblerfiles heißen DOS.I und DOSEXTENS.I.

11.2 Spezielle Datenstrukturen für Prozesse

Die folgenden Werte sind als Teil eines AmigaDOS-Prozesses zu verstehen, für jeden Prozeß ist ein kompletter Satz dieser Werte vorhanden.

Als *Prozeß* bezeichnet man einen EXEC-Task mit zusätzlichen Datenstrukturen. Eine Prozeßstruktur besteht aus:

- EXEC-Task-Struktur
- EXEC-Kanal für Meldungen
- AmigaDOS-Prozeß-Werten

Die Prozeß-Kennung, die intern von AmigaDOS verwandt wird, ist ein Pointer zum EXEC-Nachrichtenkanal.

Als Prozeß-Werte stehen zur Verfügung:

Wert	Funktion	Beschreibung
BPTR	SegArray	Feld mit den vom Prozeß benutzten SegLists
LONG	StackSize	Größe des Prozeß-Stacks in Bytes
APTR	GlobVec	Globaler Vektor für diesen Prozeß
LONG	TaskNum	CLI-Tasknummer oder 0, wenn der Prozeß kein CLI-Task ist
BPTR	StackBase	Pointer zum Ende des Prozeß-Stacks
LONG	IoErr	Wert des Sekundärergebnis des letzten Funktionsaufrufs
BPTR	CurrentDir	Lock zum aktuellen Directory
BPTR	CIS	Aktueller CLI-Eingabestrom
BPTR	COS	Aktueller CLI-Ausgabestrom
APTR	CoHand	Prozeß zur Tastaturabfrage für das aktuelle Fenster
APTR	FiHand	File-Handler-Prozeß für das aktuelle Laufwerk
BPTR	CLIStruct	Pointer zu weiteren Informationen zum CLI
APTR	ReturnAddr	Pointer zum vorhergehenden Bereich im Stack
APTR	PktWait	Funktion, die während Wartezeiten aufgerufen werden soll
APTR	WindowPtr	Pointer zum Fenster

Mit SegArray finden Sie heraus, welche Segmente ein bestimmter Prozeß belegt. Als SegArray dient ein Feld aus Langworten, dessen eigene Größe in SegArray[0] abgelegt ist. Andere Elemente sind Null oder ein BPTR zu einer SegList. Errichtet wird dieses Feld mit CreateProc. Dabei dienen die ersten beiden Elemente, die auf residenten Code zeigen, und ein drittes Element, SegList, als Argument. Wird ein Prozeß beendet, macht FreeMem den Speicher wieder frei.

StackSize zeigt die Größe des Prozeß-Stack, der dem Prozeß mit CreateProc zugewiesen wurde. Beachten Sie bitte, daß der Prozeß-Stack nicht mit dem Stack eines CLI-Befehls identisch ist. Das CLI belegt den Befehls-Stack erst kurz vor Start des Programmes. Sie können seine Größe mit der Instruktion STACK verändern. Erzeugen Sie einen neuen Prozeß, erhält AmigaDOS den Prozeß-Stack und schreibt dessen Größe nach StackSize. Die Pointer auf den Speicherplatz für den Prozeß-Kontroll-Block und auf den Stack werden ebenfalls im MemEntry-Feld der Task-Struktur gespeichert. Am Ende eines Prozesses wird der Speicherplatz durch Aufruf von FreeMem wieder verfügbar gemacht. Sie können jedoch auch eigenen Speicherplatz in die gezeigte Struktur einbinden, der wieder zur Verfügung stehen soll, wenn der Task beendet wird.

Wird ein Prozeß mit dem Aufruf von CreateProc erstellt, zeigt der Pointer GlobVec auf den Standard-Vektor des Amiga. Einige interne Prozesse benutzen jedoch besondere GlobVecs.

Der Wert von TaskNum ist normalerweise Null. Ein CLI-Prozeß speichert hier seine Ordnungszahl als Integer.

Der Pointer StackBase zeigt auf das high-memory-end des Prozeß-Stack. Dies ist das Ende des Stack, wenn in C oder Assembler programmiert wird, in Sprachen wie BCPL ist es der Anfang des Stack.

Die Werte von IoErr und CurrentDir werden von gleichnamigen AmigaDOS-Aufrufen bearbeitet. CIS und COS sind File-Handles, die die Funktionen Input und Output zurückliefern. Sie sollten diese Werte bei Programmen verwenden, die unter dem CLI laufen. Sonst sind CIS und COS Null.

CoHand und FiHand beziehen sich auf den *Console-Handler* des aktuellen Fensters und den File-Handler des aktuellen Laufwerkes. Diese Werte verwenden Sie, wenn Sie das *-Device öffnen oder ein File über den relativen Pfad-Namen suchen.

Der Pointer CLIStruct ist immer dann Null, wenn der Prozeß kein CLI-Prozeß ist. Im anderen Fall ergibt sich daraus eine Struktur, die von CLI benutzt wird. Hier das Format:

Wert	Funktion	Beschreibung
LONG	Result2	Wert von IoErr des letzten Befehls
BSTR	SetName	Name des aktuellen Directory
BPTR	CommandDir	Dem Kommando-Directory zugewiesener Lock
LONG	ReturnCode	Returncode des letzten Befehls

Wert	Funktion	Beschreibung
BSTR	CommandName	Name des aktuellen Befehls
LONG	FailLevel	Fehlerwert (gesetzt mit FAILAT)
BSTR	Prompt	aktuelles Prompt (gesetzt mit PROMPT)
BPTR	StandardIn	Voreingestelltes CLI-Eingabegerät (Terminal)
BPTR	CurrentIn	Aktuelles CLI-Eingabegerät
BSTR	CommandFile	Name des EXECUTE-Files
LONG	Interactive	Boolean; wahr, wenn Prompt benötigt
LONG	Background	Boolean; wahr, wenn CLI mit RUN gestartet
BPTR	CurrentOut	Aktuelle CLI-Ausgabe
LONG	DefaultStack	Größe des eingerichteten Stack in Langworten
BPTR	StandardOut	Voreingestelltes CLI-Ausgabegerät (Terminal)
BPTR	Module	SegList des aktuell geladenen Befehls

Die Funktion EXIT verwendet den Wert von ReturnAddr, der auf die Rücksprungadresse im Stack zeigt. Endet ein Programm durch RTS bei einem leeren Stack, springt der Rechner zu der Adresse, die CreateProc oder das CLI auf den Stack gelegt haben. Endet ein Programm mit dem Aufruf von EXIT, verwendet AmigaDOS diesen Pointer, um dieselbe Rücksprungadresse zu finden.

Der Wert von PktWait ist normalerweise Null. Ist er nicht gleich Null, ruft AmigaDOS PktWait auf, wenn ein Prozeß angehalten wird, bis ein Signal den Eingang von Daten ankündigt. Wie die Funktion GetMsg holt PktWait eine eventuell zur Verfügung stehende Information. Mit dieser Funktion werden über den *Standard-Process-Message-Port* ankommende Mitteilungen ausgefiltert, die nicht für AmigaDOS bestimmt sind.

Der Wert von WindowPtr wird verwendet, wenn AmigaDOS einen Fehler findet, der normalerweise eine Aktion des Anwenders erfordert. Ein Beispiel für einen solchen Fehler ist ein Schreibversuch auf eine schreibgeschützte oder volle Diskette. Ist der Wert von WindowPtr auf –1 gesetzt, wird der Fehler als Fehlercode von einem Open- oder Write-Aufruf an das Programm zurückgemeldet. Ist der Wert Null, erzeugt AmigaDOS einen Requester auf dem Bildschirm der Workbench, der den Anwender über den Fehler informiert und ihn auffordert, die Operation zu wiederholen oder abzubrechen. Wählt der Anwender CANCEL, übergibt AmigaDOS den Fehlercode an das Programm. Versucht er die Aktion ein zweites Mal oder legt er eine Diskette ein, beginnt AmigaDOS ein weiteres Mal mit der Ausführung.

Setzen Sie einen positiven Wert für WindowPtr ein, verwendet AmigaDOS diesen Wert als Pointer auf eine Window-Struktur. Normalerweise setzt man dafür die Struktur des gerade genutzten Fensters ein. In diesem Fall schreibt AmigaDOS die Fehlermeldung in das von Ihnen angegebene Fenster oder verwendet den Bildschirm der Workbench. Sie können den Wert des WindowPtr auch immer auf Null lassen. Verwenden Sie dann aber einen anderen

Bildschirm als den der Workbench, wird eine Fehlermeldung von Ihrem selbsterstellten Bildschirm überdeckt.

Der Ausgangswert von WindowPtr wird von dem Prozeß übernommen, der das Feld erzeugt hat. Ändern Sie den Wert von WindowPtr für ein Programm, das unter dem CLI läuft, müssen Sie den vorher aktuellen Wert zwischenspeichern und nach Beendigung Ihres Programmes wieder eintragen. Andernfalls enthält der CLI-Prozeß einen WindowPtr auf ein Fenster, das nicht mehr existiert.

11.3 Globale Datenstruktur

Diese allgemeine Datenstruktur existiert nur ein einziges Mal, wird aber von allen AmigaDOS-Prozessen benutzt. Rufen Sie die Funktion OpenLibrary auf, können Sie den *Library-Base-Pointer* (Basisadresse der Libraries) lesen. Die Basis der Datenstruktur ist ein positiver Offset vom Library Base Pointer. Der *Library-Base-Pointer* weist auf folgende Struktur:

Knotenstruktur der Libraries
APTR zum DOS RootNode
APTR zum Globalen Vektor des DOS
Für DOS reservierte Registerausgabe

Alle internen AmigaDOS-Aufrufe verwenden den Globalen Vektor als Sprungtabelle. Sie sollten ihn deshalb nicht ändern.

Die RootNode-Struktur sieht wie folgt aus:

Wert	Funktion	Beschreibung
BPTR	TaskTable	Feld mit den gerade aktiven CLI-Prozessen
BPTR	CLISegList	SegList für das CLI
LONG	Days	Tag der aktuellen Zeit
LONG	Mins	Minuten der aktuellen Zeit
LONG	Ticks	Ticks der aktuellen Zeit
BPTR	RestartSeg	SegList für den Aktualisierungsprozeß von Disketten
BPTR	Info	Pointer zu einer *Info-Substruktur*

Die TaskTable ist ein Feld, dessen Größe in TaskTable[0] gespeichert wird. Der zu dem Prozeß gehörige MsgPort jedes einzelnen CLI wird in diesem Feld gespeichert. Die Prozeßkennung für den n-ten CLI-Task ist in TaskTable[n] gespeichert. Ein nicht belegtes Feld wird mit Null beschrieben. Die Befehle NEWCLI und RUN suchen in der TaskTable nach dem nächsten Feld mit Inhalt Null und nehmen dessen TaskNum als CLI-Prozeßkennung.

Die CLISegList ist die SegList für den Code des CLI. Run und NEWCLI verwenden diesen Wert, um eine neue Instanz für das CLI zu bilden.

Im Feld RootNode ist die aktuelle Zeit gespeichert. Die Werte Days, Mins und Ticks ergeben Datum und Uhrzeit. Der Wert für Days ist die Anzahl an vergangenen Tagen seit dem 01. Januar 1978. Der Wert der Mins die Anzahl der vergangenen Minuten seit Mitternacht, ein Tick entspricht dem fünfzigsten Teil einer Sekunde. Die Zeit wird jedoch nur jede Sekunde aktualisiert.

Die RestartSeg ist die SegList für den Aktualisierungsprozeß, der immer dann aktiviert wird, wenn Sie eine neue Diskette in ein Laufwerk einlegen.

11.3.1 Die Info-Substruktur

Die nächste Unterstruktur erreichen Sie mit dem Pointer Info:

Wert	Funktion	Beschreibung
BPTR	McName	Name dieser Maschine im Netzwerk; jetzt noch Null
BPTR	DevInfo	Liste aller Devices
BPTR	Devices	Jetzt noch Null
BPTR	Handlers	Jetzt noch Null
APTR	NetHand	Prozeßkennung des Netzwerk-Handlers; jetzt noch Null

Die meisten Felder innerhalb dieser Substruktur sind zur Zeit noch leer, aber Commodore plant die Erweiterung des Systems in die Richtung auf ein lokales Netzwerk.

Die DevInfo-Struktur ist eine verkettete Liste. Mit ihr können Sie die Namen aller dem AmigaDOS bekannten Devices und Disketten feststellen. Die Einträge haben zwei verschiedene Formate, eines für Disketten, eines für alle anderen Devices. Jedem Device oder Directory (über ASSIGN definiert) entspricht diese Form des Eintrages:

Wert	Funktion	Beschreibung
BPTR	Next	Pointer zum nächsten Eintrag oder Null
LONG	Type	Zeigt den Typ des Eintrags (Device oder Directory)
APTR	Task	Handler-Prozeß oder Null
BPTR	Lock	File-System-Lock oder Null
BSTR	Handler	File-Name des Handler oder Null
LONG	StackSize	Größe des Stack für den Handler-Prozeß
LONG	Priority	Priorität des Handler-Prozesses
LONG	Startup	Dem Handler-Prozeß zugeteilter Startup-Wert
BPTR	SegList	SegList für den Handler-Prozeß oder Null
BPTR	GlobVec	Globaler Vektor für den Handler-Prozeß oder Null
BSTR	Name	Name des Device oder zugewiesener Name

Das Feld Next verkettet die Einträge, der Name des logischen Device steht im Feld Name.

Im Typen-Feld steht eine 0 (Device) oder eine 1 (Directory). Ein Directory-Eintrag wird mit ASSIGN erzeugt. Dieser Befehl weist dem angegebenen Directory einen Device-Namen zu, über den es dann angesprochen werden kann. Bezieht sich der Eintrag auf ein Directory, zeigt TASK auf den File-System-Handler der Diskette, auf dem sich das Directory befindet, das LOCK-Feld enthält einen Pointer auf den Lock dieses Directory.

Bezieht sich der Eintrag auf ein Device, kann es sich um ein residentes oder ein nicht residentes handeln. Ist es resident, identifiziert Task den Handler-Prozeß, der Lock ist normalerweise Null. Ist das Device nicht resident, ist Task Null und AmigaDOS verwendet nur den Rest der aufgezeigten Struktur.

Enthält die SegList Null, ist der Code für dieses Device nicht im Speicher. Das Feld Handler enthält einen String mit dem Namen des Files, das den Code enthält (zum Beispiel SYS:L/RAM-HANDLER). Ein Aufruf von LoadSeg liest den Code in den Speicher und schreibt das Ergebnis in das Feld SegList.

AmigaDOS erzeugt dann einen neuen Handler-Prozeß mit den Werten von SegList, StackSize und Pri. Der neue Prozeß ist ein BCPL-Prozeß und benötigt einen globalen Vektor, das ist der Wert, den Sie mit GlobVec angegeben haben, oder einen neuen, eigenen globalen Vektor, falls GlobVec Null ist.

Der neue Prozeß erhält dann sofort eine Nachricht, die den ursprünglich vergebenen Namen, den in Startup gespeicherten Wert und den Beginn des Eintrags in der Liste enthält. Der neue Handler-Prozeß schreibt dann die Prozeßkennung in ein leeres Feld von TaskTable, sofern dies nötig ist. Ist dieser Eintrag gemacht, weisen alle weiteren Referenzen zu diesen Device-Namen und verwenden denselben Handler-Task. Dieser Ablauf trifft zum Beispiel auf das Device RAM: zu. Wird in der TaskTable kein Eintrag vorgenommen, führen weitere Referenzen zu diesem Device immer zu neuen Device-Handler-Prozessen. Diese Aktionen laufen für das Device CON: ab.

Ist der Eintrag im Typen-Feld gleich 2 (dt_volume), sieht das Format der List-Struktur wie folgt aus:

Wert	Funktion	Beschreibung
BPTR	Next	Pointer zum nächsten Eintrag oder Null
LONG	Type	Zeigt den Typ des Eintrags (Volume)
APTR	Task	Handler-Prozeß oder Null
BPTR	Lock	File-System-Lock oder Null
LONG	VolDays	Zeitpunkt der Erstellung des Volume
LONG	VolMins	
LONG	VolTicks	
BPTR	LockList	Liste der aktiven Locks dieses Volume
LONG	DiskType	Art der Diskette
LONG	Spare	Nicht benutzt
BSTR	Name	Name des Volume

In diesem Fall steht der Name des Volume im Feld Name. Das Task-Feld verweist auf den Handler-Prozeß, wenn das Volume eingelegt ist, oder hat den Wert Null. Um gleichnamige Disketten zu unterscheiden, trägt AmigaDOS den Zeitpunkt der Formatierung in die List-Struktur ein. Bei Bedarf liest AmigaDOS bei gleichnamigen Disketten diesen *Zeitstempel* aus.

Ist eine Diskette gerade nicht in ein Laufwerk eingelegt, werden die aktuellen Locks im LockList-Feld gespeichert. AmigaDOS verwendet das DiskType-Feld zur Identifizierung des Diskettentyps. Normalerweise ist dies eine AmigaDOS-Diskette. Der Diskettentyp besteht aus maximal vier Zeichen, die höherwertigen Bits des Langwortes werden mit Null aufgefüllt.

11.4 Speicherverwaltung

AmigaDOS verwaltet den gesamten Arbeitsspeicher mit Hilfe der von Exec bereitgehaltenen Funktion AllocMem. Auf diesem Weg werden unter anderem sämtliche Strukturen und File-Handler errichtet. Nicht mehr benötigter Speicher wird mit Aufruf der Funktion FreeMem wieder zur Verfügung gestellt. Jedes von AmigaDOS errichtete Speichersegment enthält einen BPTR auf das zweite Langwort in dieser Struktur. Das erste Langwort enthält immer die Länge des gesamten Datenblocks in Bytes. Die Struktur eines reservierten Speicherbereichs sieht also immer so aus:

Wert	Funktion	Beschreibung
LONG	BlockSize	Größe des Speicherblocks in Byte
LONG	FirstData	Datenbeginn. Ein BPTR zeigt auf diese Adresse.

11.5 Segment-Listen

Zum Erzeugen einer Segmentliste wird SegList aufgerufen. Als Ergebnis wird ein BPTR auf einen neu reservierten Speicherblock zurückgeliefert. Die Länge jedes der in der Liste aufgeführten Blöcke ist in dem Langwort bei der Adresse BPTR-4 abzulesen. Die Gesamtlänge des Blocks ist dann der ausgelesene Wert plus 8 (vier für das Link-Feld und vier für das Size-Feld selbst).

Die SegList ist eine Liste, die durch BPTRs zusammengebunden wird. Der letzte Block enthält statt des Pointers auf den nächsten eine Null. Der Rest jedes Eintrages der SegList enthält den geladenen Code:

Wert	Funktion	Beschreibung
LONG	NextSeg	BPTR zum nächsten Segment oder Null
LONG	FirstCode	Erster Wert des Binärfiles

11.6 File-Handles

File-Handles werden von der AmigaDOS-Funktion Open erzeugt. Sie werden unter anderem als Argument für die Funktionen Read und Write benötigt. AmigaDOS liefert einen File-Handle als einen BPTR auf folgende Struktur zurück:

Wert	Funktion	Beschreibung
LONG	Link	Nicht benutzt
LONG	Interact	Boolean, WAHR wenn es um ein interaktives File ist
LONG	ProcessID	Prozeßkennung des Prozeß-Handler
BPTR	Buffer	Puffer für internen Gebrauch
LONG	CharPos	Zeichenposition für internen Gebrauch
LONG	BufEnd	Endposition für internen Gebrauch
APTR	ReadFunc	Funktion zum Auslesen des Puffers
APTR	WriteFunc	Funktion wird aufgerufen, wenn Puffer voll ist
APTR	CloseFunc	Funktion wird aufgerufen, wenn Handle geschlossen wird
LONG	Arg1	Argument; abhängig vom Typ des File-Handle
LONG	Arg2	Argument; abhängig vom Typ des File-Handle

Die meisten dieser Felder werden von AmigaDOS intern benutzt und sollten deshalb nicht modifiziert werden; lediglich das erste Feld kann zum Zusammenbinden von File-Handles zu einer Liste dienen. Normale Operationen verwenden den Pointer auf ein solches File-Handle nur als Parameter und kümmern sich nicht um die interne Struktur.

Diese Beschreibung paßt nicht zu DOSEXTENS.H oder DOSEXTENS.I. Verwenden Sie vorsichtshalber die Information aus den Include-Files.

11.7 Locks

Das AmigaDOS-Filesystem nutzt eine spezielle Datenstruktur, den *Lock*, sehr intensiv. Diese Struktur erledigt zwei Aufgaben. Eine Funktion ist die Koordination eines oder mehrerer Lesezugriffe oder eines Schreibzugriffs auf ein File. Beachten Sie bitte, daß ein solcher *shared lock* nicht die Aktualisierung eines Directory verhindert.

Zum zweiten stellt ein Lock eine einheitliche Identifizierung für ein File dar. Ein File kann auf vielen Wegen angesprochen werden, der Lock ist ein einfacher Handle zu diesem File. Der Lock enthält den Diskettenblock, in dem sich das Directory oder der File-Header befinden.

Die Struktur des Lock sieht wie folgt aus:

Wert	Funktion	Beschreibung
BPTR	NextLock	BPTR zum nächsten Lock der Kette oder Null
LONG	DiskBlock	Blocknummer des Directory oder File-Header
LONG	AccessType	Geteilter (shared) oder ausschließlicher Zugriff
APTR	ProcessID	Prozeßkennung des Handler-Task
BPTR	VolNode	Eintrag im Volume für diesen Lock

Da AmigaDOS das Feld NextLock verwendet, um Locks aneinanderzuketten, sollten Sie es nicht verändern. Das File-System verwendet das DiskBlock-Feld, um den Block auf der Diskette einzutragen, in dem das Directory oder der File-Header stehen. Das Feld AccessTyp enthält Informationen, ob es sich bei dem Lock um einen *shared-read-Lock* (–2) oder einen *exclusive-write-Lock* (–1) handelt. Das Feld ProcessID enthält einen Pointer zu dem Handler-Prozeß des Device, auf dem das File zu finden ist, auf das sich der jeweilige Lock bezieht. Das VolNode-Feld letztlich verweist auf den Knoten in der DevInfo-Struktur, der das Volume identifiziert, auf das sich der Lock bezieht. Einträge zu einem Volume in der DevInfo-Struktur enthalten Informationen, ob die Diskette eingelegt ist oder Locks auf dieses Volume offen sind.

Beachten Sie bitte, daß ein Lock auch Null sein kann. In diesem besonderen Fall zeigt ein Lock auf den Ursprung des initialen File-Systems, das FiHand-Feld innerhalb der Prozeß-Daten-Struktur gibt Auskunft über den Handler-Prozeß.

11.8 Packets

Über *Packets* läuft die gesamte Kommunikation zwischen den Prozessen. Ein Packet ist die Struktur, die dem Mechanismus der Datenübertragung zwischen Prozessen, den Exec zur Verfügung stellt, aufbaut.

Die Exec-Mitteilung ist eine an anderer Stelle erklärte Struktur, die als einen Bestandteil ein Name-Feld hat. AmigaDOS verwendet dieses Feld als APTR zu einer anderen Struktur im Speicher, Packet genannt. Ein Packet muß Langwort-ausgerichtet sein und hat folgende Struktur:

Wert	Funktion	Beschreibung
APTR	MsgPtr	Pointer zurück zur Message-Struktur
APTR	MsgPort	Message-Port, an den die Antwort gesandt werden soll
LONG	PktType	Packet-Typ
LONG	Res1	Erstes Ergebnis-Feld
LONG	Res2	Zweites Ergebnis-Feld
LONG	Arg1	Argument 1; abhängig vom Packet-Typ

Wert	**Funktion**	**Beschreibung**
LONG	Arg2	Argument 2; abhängig vom Packet-Typ
	......	
LONG	ArgN	Argument N; abhängig vom Packet-Typ

Das genaue Format eines Packets hängt stark von seinem Typ ab; in allen Fällen aber enthält es einen Pointer zurück zur Nachricht, den MsgPtr, und zwei Ergebnis-Felder. Sendet AmigaDOS ein Packet, wird der MsgPtr mit der Prozeß-Kennung des Absenders überschrieben, so daß das Ergebnis an den richtigen Empfänger gesendet werden kann. Lediglich bei der Sendung an einen AmigaDOS-Handler-Prozeß müssen Sie den Wert des Absenders selbst in den MsgPtr schreiben, da AmigaDOS den Port überschreibt, wenn die Meldung zurückkommt. Außer dem Ergebnis-Feld werden alle anderen Felder von AmigaDOS verwaltet.

Alle Packets werden an den Message-Port gesandt, der bei der Erzeugung des Prozesses eingerichtet wird. Dieser ist immer empfangsbereit. Ankommende Nachrichten setzen das Signal 8. Wenn dieser Prozeß dann aktiv wird (auf Nachrichten wartende Prozesse warten darauf, daß Signal 8 gesetzt wird), übernimmt GetMsg die Meldung vom MsgPort und filtert die Packet-Adresse heraus. Ist der Prozeß vom Typ *AmigaDOS-Handler-Prozeß*, enthält das Packet einen Wert im PktType, der eine Aktion ausführt, zum Beispiel Daten liest. Die Argument-Felder enthalten spezielle Werte, zum Beispiel die Adresse und Größe des Puffers, in den die Zeichen geschrieben werden.

Hat der Handler-Prozeß diese Aufgabe abgeschlossen, wird die Meldung im gleichen Format zum Sender zurückgeschrieben. Die Mitteilungs-Struktur und die Packet-Struktur müssen dem Empfänger zugeteilt und dürfen nicht freigegeben werden, bevor die Rückmeldung erfolgt. Normalerweise wird AmigaDOS vom Empfänger aufgefordert, das Packet zu senden, wenn eine Aufforderung zum Lesen erteilt wurde. In einigen Fällen jedoch, zum Beispiel wenn asynchrone Datenübertragung definiert wurde, sendet der Handler-Prozeß selbst die erforderlichen Packets. Die Mitteilungs-Struktur und die Packet-Struktur müssen dem Empfänger zugeteilt werden, und das Feld mit der Prozeßkennung wird mit dem Wert des MsgPort des Senders, zu dem die Meldung zurücklaufen soll, beschrieben. Beachten Sie bitte, daß viele Packets an mehrere Empfänger geschickt werden können, der Rücklauf jedoch immer zum selben Sender geht.

Unter DOS 1.2 gibt es einen neuen *Packet-Action-Typ*: SetFileDate (34). Sie können diesen Packet-Typ dazu verwenden, das File- oder Directory-Datum auf einen bestimmten Wert zu setzen. Das erste Argument ist ein Lock, das zweite ein APTR auf ein AmigaDOS-Datum, das durch den Aufruf der DateStamp-Routine zurückgegeben wird.

Ein weiterer neuer Packet-Action-Typ ist SetRawMode (994). Dieser neue Modus wird dazu verwendet, das CON:-Device so zu ändern, daß es genauso reagiert wie das Device RAW:, beziehungsweise um es wieder in den ursprünglichen Zustand zurückzusetzen. Übergeben

Sie das TRUE-Argument, um in den RAW-Zustand zu schalten. FALSE stellt den normalen Zustand ein.

Andere Packets umfassen Flush (27) und MoreCache (18).

Trotzdem existiert ein Unterschied zwischen RAW: und CON: im RAW-Zustand. Sowohl im normalen als auch im RAW-Zustand akzeptiert CON: die Escape-Sequenzen zum Ein- oder Ausschalten der Umwandlung von Linefeed in Carriage Return plus Linefeed. RAW: dagegen formt niemals Linefeeds um (die Sequenzen lauten: CSI 20h zum Einschalten der Konvertierung und CSI 201 zum Ausschalten).

11.9 Packet-Typen

AmigaDOS unterstützt die im folgenden beschriebenen Packet-Typen. Nicht alle Typen sind jedoch für alle Handler zulässig, zum Beispiel sind Zugriffe mit Rename nur für Handler zulässig, die ein Filing-System unterstützen. Für jeden Packet-Typ werden die Argumente und Ergebnisse beschrieben. Der aktuelle Code für jeden Typ erscheint in dezimaler Form unmittelbar neben dem Namen des Symbols. In fast allen Fällen wird ein Fehler durch eine Null im Feld Res1 angezeigt. Dann erscheinen im Feld Res2 zusätzliche Informationen über diesen Fehler. Mit dem Aufruf der Funktion IoErr aus AmigaDOS werden diese zusätzlichen Informationen ausgegeben.

Open Old File

Typ	LONG	Action.FindInput (1005)
Arg1	BPTR	FileHandle
Arg2	BPRT	Lock
Arg3	BSTR	Name
Res1	LONG	Boolean

Die Funktion öffnet ein bereits bestehendes File für Ein- oder Ausgabe (im *AmigaDOS-Programmierer-Handbuch* finden Sie unter Open mehr Informationen über das Öffnen von Files). Um den Lock zu erhalten, müssen Sie DeviceProc aufrufen, um die Handler-Prozeßkennung festzustellen. IoErr liefert dann den gewünschten Lock. Alternativ dazu können Sie diese Werte auch direkt aus der DevInfo-Struktur lesen. Beachten Sie bitte, daß sich der Lock auf das Directory bezieht, in dem das File steht, nicht auf das File selbst.

FileHandle muß vom aufrufenden Programm bereitgestellt und initialisiert werden. Dazu werden alle Felder außer CharPos und BufEnd auf Null gesetzt. Die beiden Ausnahmen erhalten den Wert –1. Das Feld ProcessID im File-Handle muß den Wert der Prozeßkennung des Handler-Prozesses tragen.

Das Ergebnis des Aufrufs ist Null, wenn ein Fehler aufgetreten ist. In diesem Fall stehen im Feld Res2 weitere Informationen zu dem aufgetretenen Fehler zur Verfügung. Der File-Handle sollte dann gelöscht werden.

Open New File

Typ	LONG	Action.FindOutput (1006)
Arg1	BPTR	FileHandle
Arg2	BPTR	Lock
Arg3	BSTR	Name
Res1	LONG	Boolean

Die Argumente von Open New File sind gleich denen von Open Old File.

Read

Typ	LONG	Action.Read (82)
Arg1	BPTR	FileHandle Arg1
Arg2	APTR	Puffer
Arg3	BSTR	Länge
Res1	LONG	Aktuelle Länge

Zum Lesen eines Files wird die Prozeßkennung aus dem Feld ProcessID des File-Handle entnommen. Und der Wert des Feldes Arg1 vom Handle wird in das Feld Arg1 des Packets übernommen. In die beiden anderen Argument-Felder werden die Adresse und Länge des Puffers geschrieben. Im Ergebnis-Feld steht dann die Anzahl der gelesenen Zeichen. Bei der Erläuterung zur Funktion Read finden Sie weitere Informationen zu diesem Thema. Tritt ein Fehler auf, wird bei diesem Aufruf –1 gemeldet, auch hier stehen im Feld Res2 weitere Informationen zu dem aufgetretenen Fehler zur Verfügung.

Write

Typ	LONG	Action.Write (87)
Arg1	BPTR	FileHandle Arg1
Arg2	APTR	Puffer
Arg3	BSTR	Länge
Res1	LONG	Aktuelle Länge

Die Argumente sind identisch mit denen von Read. Weitere Informationen zum Feld Res1 finden Sie bei der Beschreibung der Funktion READ.

Close

Typ	LONG	Action.End (1007)
Arg1	BPTR	FileHandle Arg1
Res1	LONG	WAHR

Mit diesem Packet wird ein geöffneter File-Handle geschlossen. Die Prozeßkennung wird vom File-Handle übernommen. Die Funktion hat normalerweise das Ergebnis WAHR.

Nachdem ein File-Handle geschlossen wurde, sollte der dadurch nicht mehr benötigte Speicherplatz wieder verfügbar gemacht werden.

Seek

Typ	LONG	Action.Seek (1008)
Arg1	BPTR	FileHandle Arg1
Arg2	LONG	Position
Arg3	LONG	Modus
Res1	LONG	Alte Position

Dieses Packet unterstützt den Aufruf von Seek. Als Ergebnis wird die alte Position oder –1 gemeldet, wenn ein Fehler auftritt. Die Prozeßkennung wird vom File-Handle übernommen.

WaitChar

Typ	LONG	Action.WaitChar (20)
Arg1	LONG	Timeout
Res1	LONG	Boolean

Dieses Packet implementiert die Funktion WaitForChar. Sie müssen es zum Console-Handler-Prozeß senden, mit der zu wartenden in Arg1. Das Packet meldet sich zurück, wenn Zeichen zur Übernahme anstehen oder wenn die angegebene Zeit abgelaufen ist. Wird als Ergebnis WAHR übergeben, wurde mindestens ein Zeichen von einem nachgeordneten READ gelesen.

ExamineObject

Typ	LONG	Action.ExamineObject (23)
Arg1	BPTR	Lock
Arg2	BSTR	FileInfoBlock
Res1	LONG	Boolean

Dieser Packet-Typ implementiert die Examine-Funktion. Er filtert die Prozeßkennung des Handlers aus dem Feld ProcessID des Lock. Ist der Lock Null, wird der normale File-Handler verwendet, der im Feld FiHand des Prozesses abgelegt ist. Das Ergebnis ist Null, wenn ein Fehler auftritt, weitere Informationen über die Fehlerursache stehen dann in Res2. Der FileInfoBlock enthält ansonsten den Inhalt der Felder Name und Kommentar als BSTR.

ExamineNext

Typ	LONG	Action.ExamineNext (24)
Arg1	BPTR	Lock
Arg2	BPTR	FileInfoBlock
Res1	LONG	Boolean

Dieser Aufruf implementiert die Funktion ExNext, die Argumente entsprechen denen von Examine. Beachten Sie, daß der BSTR, der den Filenamen enthält, zwischen den Aufrufen

von ExamineObject und ExamineNext nicht gelöscht werden darf, da er den Namen nur als Platzhalter innerhalb des zu untersuchenden Directory verwendet.

DiskInfo

Typ	LONG	Action.DiskInfo (25)
Arg1	BPTR	InfoData
Res1	LONG	WAHR

Mit diesem Packet wird die Funktion Info realisiert. Normalerweise liefert der Lock zu einem Device auch die Prozeßkennung für den Handler. Dieses Packet kann ebenso zu einem Console-Handler-Prozeß gesandt werden. In diesem Fall enthält das Volume-Feld in InfoData den Pointer zu dem Fenster, das der Console-Handler geöffnet hat.

Parent

Typ	LONG	Action.DiskInfo (25)
Arg1	BPTR	InfoDaten
Res1	LONG	WAHR

Dieses Packet meldet als Ergebnis den Lock, aus dem der in Arg1 angegebene Lock errichtet wurde. Es entspricht dem ParentDir-Funktionsaufruf. Dazu wird die Prozeßkennung des Handlers aus dem Feld ProcessID des Lock herausgefiltert. Ist der Lock Null, wird der normale File-Handler verwendet, der im Feld FiHand des Prozesses abgelegt ist.

DeleteObject

Typ	LONG	Action.DiskInfo (25)
Arg1	BPTR	InfoDaten
Res1	LONG	WAHR

Dieses Packet implementiert die Funktion Delete. Es erhält den dazu benötigten Lock vom Aufruf der Funktion IoErr, der unmittelbar auf einen erfolgreichen Aufruf der Funktion DeviceProc folgt. Aus DeviceProc wird die Prozeßkennung übernommen. Der aktuelle Lock gehört zu dem Directory, in dem das zu löschende File abgelegt ist, so wie es oben erklärt wird.

CreateDir

Typ	LONG	Action.CreateDir (22)
Arg1	BPTR	Lock
Arg2	BSTR	Name
Res1	LONG	Lock

Dieses Packet implementiert die Funktion CreateDir. Die Argumente entsprechen denen von DeleteObject. Das Ergebnis ist Null (im Fehlerfall) oder der Lock des neuen Directory.

LocateObject

Typ	LONG	Action.LocateObject (8)
Arg1	BPTR	Lock
Arg2	BSTR	Name
Arg3	LONG	Modus
Res1	BPTR	Lock

Dieses Packet implementiert die Funktion Lock und liefert Null oder den gewünschten Lock. Die Argumente entsprechen denen von CreateDir, dazu kommt der Modus als Argument 3. Der Modus bezieht sich auf den Typ des Lock, also geteilter oder exklusiver Zugriff.

CopyDir

Typ	LONG	Action.CopyDir (19)
Arg1	BPTR	Lock
Res1	BPTR	Lock

Dieses Packet implementiert die Funktion DupLoc. Ist der Lock, der dupliziert werden soll, gleich Null, ist auch das Duplikat Null. Andernfalls wird die Prozeßkennung aus dem Lock gefiltert und dieser Packet-Typ gesendet. Das Ergebnis ist der neue Lock oder Null, wenn ein Fehler aufgetreten ist.

FreeLock

Typ	LONG	Action.FreeLock (15)
Arg1	BPTR	Lock
Res1	LONG	Boolean

Dieser Aufruf implementiert die UnLock-Funktion. Er filtert die Prozeßkennung aus dem Lock. Beachten Sie, daß die Funktion UnLock auf einen Lock mit dem Wert Null nichts bewirkt.

SetProtect

Typ	LONG	Action.SetProtect (21)
Arg1	nicht verwendet	
Arg2	BPTR	Lock
Arg3	BSTR	Name
Arg4	LONG	Maske
Res1	LONG	Boolean

Dieses Packet implementiert die SetProtect-Funktion. Der Lock ist ein Lock zu dem Directory, das von DeviceProc geliefert wird, wie es schon oben bei DeleteObject beschrieben wurde. Die letzten vier Bit von Maske repräsentieren die Zugriffe Read, Write, Execute und Delete in dieser Reihenfolge. (Delete entspricht also Bit Null.)

SetComment

Typ	LONG	Action.SetComment (28)
Arg1	nicht verwendet	
Arg2	BPTR	Lock
Arg3	BSTR	Name
Arg4	BSTR	Kommentar
Res1	LONG	Boolean

Mit diesem Packet-Typ wird die SetComment-Funktion implementiert. Die Argumente entsprechen denen von SetProtect, lediglich in Arg4 ist der zu vergebende Kommentar als BSTR enthalten.

RenameObject

Typ	LONG	Action.RenameObject (17)
Arg1	BPTR	vom Lock
Arg2	BPTR	vom Namen
Arg3	BPTR	zum Lock
Arg4	BPTR	zum Namen
Res1	LONG	Boolean

Dieses Packet implementiert die Funktion RENAME. Es benötigt dazu einen Lock zum Directory und den ursprünglichen sowie den neuen Namen. Das Directory wird von DeviceProc geliefert, wie oben bei DeleteObject beschrieben wurde.

Inhibit

Typ	LONG	Action.Inhibit (31)
Arg1	LONG	Boolean
Res1	LONG	Boolean

Dieses Packet implementiert eine Funktion, die nicht als AmigaDOS-Funktion zur Verfügung steht. Das Packet enthält einen Wahrheits-Wert, der besagt, ob die Dateiverwaltung am Verifizieren einer neu eingelegten Diskette gehindert werden soll. Ist der Wert WAHR, können Sie Disketten wechseln, ohne daß die Dateiverwaltung den Versuch unternimmt, die Diskette zu verifizieren. Während das Wechseln von Disketten für den Computer nicht erkennbar ist, ist der Disketten-Typ mit dem Wert Not a DOS disk (keine DOS-Diskette) belegt, andere Prozesse können dann nicht auf die Diskette zugreifen.

Ist der Wert FALSCH, kehrt das System zur normalen Routine zurück, sobald die Verify-Operation bei der zuvor eingelegten Diskette beendet ist.

Diese Funktion ist besonders dann sinnvoll, wenn Sie ein Programm wie DISKCOPY schreiben, bei dem viele Diskettenwechsel vorgenommen werden, dabei aber eine Diskette mit unvollständiger Struktur verwendet wird. Nach Aufruf der Funktion bleiben Ihnen die

vielen Fehlermeldungen des *Disk-Validator* erspart, der Sie sonst immer auf die unvollständige Diskette hinweist.

RenameDisk

Typ	LONG	Action.RenameDisk (9)
Arg1	BPTR	neuer Name
Res1	LONG	Boolean

Auch diese Funktion ist nicht als normaler AmigaDOS-Aufruf verfügbar. Das einzige Argument enthält den neuen Namen der Diskette, die gerade im aktuellen Laufwerk des Drive-Handlers eingelegt ist, zu dessen Filesystem-Prozeß das Packet gesandt wurde. Der Diskettenname wird auf der Diskette und im Speicher geändert.

Kapitel 12: Weiterführende Hinweise für den fortgeschrittenen Entwickler

Dieses Kapitel greift verschiedene Themen auf, die vor allem den fortgeschrittenen Softwareentwickler ansprechen. So wird auf die Einrichtung neuer Devices ebenso eingegangen wie auf die Programmierung eines Amiga, dessen Hauptspeicher über 512 Kbyte hinaus erweitert wurde.

Im einzelnen werden besprochen:

Der Aufbau eines Overlay-Moduls
zum Zusammenschluß sehr umfangreicher Programme.

Das ATOM-Hilfsprogramm
arbeitet auf einem neuen binären Fileformat und erlaubt dem Entwickler, die passenden Load-Bits zu setzen. Achten Sie darauf, daß der Programmcode und alle Daten im CHIP-Speicher, den ersten 512 Kbyte stehen, da das Programm nur diesen Speicher berücksichtigt. Andernfalls funktioniert das Programm auf einer Maschine mit mehr Arbeitsspeicher nicht.

Das Einbinden eines neuen DISK-Device in AmigaDOS
Das Einbinden einer Hard-Disk oder eines weiteren Laufwerkes in den adressierbaren Teil des Filing-Systems.

Das Einbinden eines neuen Device (kein Diskettenlaufwerk) in AmigaDOS
Das Einbinden von Geräten wie einem weiteren seriellen oder parallelen Port, einer zusätzlichen RAM-Disk oder eines Grafik-Tabletts in AmigaDOS. (Keine Laufwerke !)

Die Benutzung von AmigaDOS ohne Intuition
für Programmierer, die auf das Bildschirm-Handling verzichten wollen, das von Intuition zur Verfügung gestellt wird.

12.1 Überblick über die Modul-Overlay-Tabelle

Werden Overlays verwendet, erzeugt der Linker zuerst ein langes File aus allen Objektfiles, das eine Folge von Modulen mit relozierbarem Code enthält. Die Modul-Overlay-Tabelle enthält eine Datenstruktur, die jedes Modul und deren Beziehung zueinander beschreibt.

Erstellen Sie ein Programm, das Overlays verwenden soll, müssen Sie sich vor Augen halten, wie der Overlay-Supervisor die Interaktionen zwischen den einzelnen Segmenten steuert. Zuerst sollten Sie den Baum erstellen, aus dem hervorgeht, in welcher Beziehung die einzelnen Module zueinander stehen. Diese Beziehungen teilen Sie dann dem Linker mit.

Die Modul-Overlay-Tabelle besteht aus acht Langworten, jedes von ihnen beschreibt einen Overlay-Knoten und ist Teil des Overlay-Files. Jeder Acht-Wort-Eintrag enthält folgende Daten:

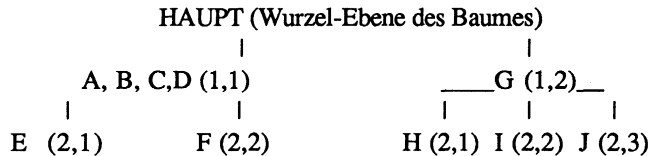
Wert	Funktion	Beschreibung
LONG	seekOffset;	/* Wo im File befindet sich dieser Knoten */
LONG	dummy1;	/* Ein Wert von 0 ... zusammengehöriger Punkte */
LONG	dummy2;	/* Ein Wert von 0 ... zusammengehöriger Punkte */
LONG	level;	/* Ebene im Baum */
LONG	ordinate;	/* Anzahl an Punkten auf dieser Ebene */
LONG	firstHunk;	/* Modul-Nummer des ersten Knotens zu diesem Modul */
LONG	symbolHunk;	/* die Nummer des Moduls, zu dem dieses Symbol gehört */
LONG	symbolOffsetX;	/* (Offset+4), wobei Offset der Offset des Symbol-Hunk
		/* zu dem jeweiligen Symbol-Eintrag ist. */

Jeder dieser Einträge wird in den nun folgenden Abschnitten näher beschrieben.

12.1.1 Einen Overlay-Baum planen

Gehen wir davon aus, daß die folgenden Module an der Overlay-Struktur beteiligt sind: HAUPT, A, B, C, D, E, F, G, H, I und J. HAUPT ruft die Module A, B, C und D auf, und jedes dieser Module kann HAUPT aufrufen. Weiterhin nehmen wir an, daß Modul E von den Modulen HAUPT, A, B und C aufgerufen werden kann, aber keine Beziehung zum Modul F hat. Soll also eine Routine aus E aufgerufen werden, müssen weiterhin die Module A, B, C und D im Speicher zur Verfügung stehen. Routine F verhält sich wie E, das heißt, alle Module außer E müssen im Speicher bereitstehen, wenn F von A, B, C oder D aufgerufen wird. Der Overlay-Supervisor kann also die Routinen E und F nacheinander in den gleichen Bereich des Speichers schreiben, da die eine die andere zur Funktion nicht benötigt.

Nehmen wir nun weiter an, Modul G belegt denselben Speicherbereich wie A, B, C und D und auch die Routinen H, I und J passen in denselben Platz, so ergibt sich folgende Struktur für die Basis unseres Baumes:



Nun haben wir nicht nur den Baum aufgezeichnet, sondern auch gleich noch seine Äste benannt. Diese Modul-Overlay-Nummern finden sich auch in der Modul-Overlay-Tabelle und bezeichnen die Knoten, denen sie zugeordnet sind.

Aus der Beschreibung können Sie ersehen, daß alle Routinen aus den Modulen A, B, C und D immer gleichzeitig im Speicher stehen, wenn aus HAUPT eine dieser Routinen aufgerufen wird, da sie vom Linker alle im gleichen Knoten untergebracht sind. Während A–D im Speicher stehen, kann der Linker File E immer gleich einlesen und reinitialisieren (bei jedem Aufruf neu!), wenn eine Routine daraus aufgerufen wird. Wird aus einem der Files A–D das Modul F benötigt, ersetzt der Linker den Inhalt des Speichers, in dem E steht, durch F und initialisiert dieses Modul. So sind die Files A–D die erste Ebene im Overlay-Baum. Die Routinen in E und F stellen Ebene 2 dar, alle Module aus Ebene 1 müssen im Speicher stehen, um sie verfügbar zu machen.

Eine Routine kann natürlich nur dann auf andere Module zugreifen, wenn dieses andere Modul im Speicher steht oder im direkt nachgeordneten Knoten eingereiht ist. HAUPT kann also nicht auf E zugreifen, ohne vorher A–D eingelesen zu haben. E kann jedoch auf Routinen in HAUPT zugreifen, da HAUPT immer im Speicher steht, wenn E aufgerufen wird.

Beachten Sie auch, daß jeder Hauptast und seine Knoten wieder mit 1 beginnend numeriert werden.

12.1.2 Beschreibung des Baums

Der Baum wird erzeugt, indem Sie dem Linker dessen Struktur mitteilen. Die numerischen Werte, gleich denen in unserer Zeichnung oben, werden vom Linker nacheinander zugeteilt und erscheinen in der Modul-Node-Tabelle.

Hier nun die Sequenz, die den oben gezeigten Baum entstehen läßt:

```
OVERLAY
a,b,c,d
*e
*f
g
*h
*i
*j
```

Diese Beschreibung zeigt dem Linker, daß A, B, C und D Teile eines Knotens in einer gegebenen Ebene (in diesem Fall Ebene 1) sind. Der Stern (*) vor E und F zeigt an, daß diese Module eine Ebene unter A–D liegen. Sie können also nur durch A–D oder von im Baum weiter oben stehenden Module aufgerufen werden. Der Name G trägt keinen Stern, damit steht er auf derselben Ebene wie A–D. Somit erfährt der Linker, daß entweder A–D oder G im Speicher sein können, nicht jedoch alle Files. H–J stehen eine Ebene unter G.

Die obenstehenden Abschnitte haben den Ursprung des Overlay-Baumes und die Reihenfolge der ersten Files in der Modul-Overlay-Symbol-Tabelle erklärt.

seekOffset
Der erste Wert für jeden Knoten in der Overlay-Tabelle ist der *seekOffset*. Wie bereits gesagt, bildet der Linker aus allen Overlay-Knoten ein großes File. Der Wert des seekOffset wird der seek-Routine übergeben (mit den Parametern file, byte_offset) und ergibt den Pointer auf das erste Byte des hunkHeader des Knotens.

initialHunk
Der Wert in *initialHunk* wird vom Overlay-Supervisor verwendet, wenn ein Knoten überschrieben wird. Er verweist auf das Modul, das geladen sein muß, um den Knoten zu laden, der das Symbol enthält. Wird eine Routine aufgerufen, die zu einem anderen Knoten oder einer anderen Ebene gehört (wenn es sich nicht um ein direkt dazugehöriges, ein direkt nachgeordnetes oder ein direkt davorstehendes File handelt), muß der Speicher freigemacht werden. Dazu werden alle nicht benötigten Module gelöscht und Raum für das Modul mit dem angegebenen Symbol gemacht.

SymbolHunk und SymbolOffsetX
Diese Einträge zu den Symbolen werden vom Overlay-Supervisor verwendet, um den aktuellen Einsprungpunkt eines geladenen oder zu ladenden Moduls zu finden. SymbolHunk zeigt auf das Modul, in dem das Symbol zu finden ist, SymbolOffsetX-4 zeigt den Offset vom Beginn des Moduls an, bei dem der Einsprungpunkt im Augenblick liegt.

12.2 ATOM (Alink Temporary Object Modifier)

Dieser Abschnitt erklärt das ATOM-Hilfsprogramm, seine Entstehungsgeschichte, die Art der Implementierung und Alternativen zu seinem Einsatz.

12.2.1 Das »Problem«

Programmierer müssen in der Lage sein, zu bestimmen, daß Teile ihres Programmes in den Chip-RAM (die ersten 512 Kbyte des RAM) geladen werden, damit die Custom-Chips darauf zugreifen können. Sie möchten aber auch, daß diese speziellen Daten wie andere Daten behandelt werden können, daher müssen sie normal eingebunden und geladen werden.

12.2.2 Die bisherige Lösung

Der bisher benutzte Weg zur Lösung dieses Problems sah so aus: Zuerst wurde AllocMem mit gesetzten Chip-RAM-Bits aufgerufen. Dann wurden die Daten von der Speicherstelle, in die sie geladen wurden, in den Chip-RAM kopiert. Auf diese Weise waren die Daten zweimal im Speicher vorhanden: einmal die ursprünglich geladenen und dann die in den ersten 512 Kbyte des Speichers.

Als andere Lösung blieb nur, dem Anwender zu sagen, daß das Programm in Rechnern mit mehr als 512 Kbyte nicht läuft. Aber das ist eine ziemlich unbefriedigende Lösung.

12.2.3 Die ATOM-Lösung

1. Kompilieren oder assemblieren Sie wie gewohnt.
2. Bearbeiten Sie das Object-File mit dem Post-Prozessor ATOM. ATOM arbeitet interaktiv mit dem Anwender und fragt jedes Modul des Object-Files ab. Alle angegebenen werden im Typ als für CHIP-Speicher geändert.
3. Der Linker kennt nun neben den schon bekannten Modul-Typen hunk_code, hunk_data und hunk_bss noch sechs weitere Modul-Typen:

hunk_code_chip	=	hunk_code	+ Bit 30 gesetzt
hunk_code_fast	=	hunk_code	+ Bit 31 gesetzt
hunk_data_chip	=	hunk_data	+ Bit 30 gesetzt
hunk_data_fast	=	hunk_data	+ Bit 31 gesetzt
hunk_bss_chip	=	hunk_bss	+ Bit 30 gesetzt
hunk_bss_fast	=	hunk_bss	+ Bit 31 gesetzt

   Der Linker bereitet alle Modul-Typen zur Bearbeitung mit dem Loader vor (und bindet sie zusammen, falls erforderlich). Der Loader verwendet die Informationen aus dem Modul-Header während des Ladens.

Beachten Sie, daß hunk_code ausführbare Maschinensprache-Elemente, hunk_data initialisierte Daten und hunk_bss nicht initialisierte Daten (Felder, Variablendefinitionen und so weiter) enthalten.

4. Der Loader lädt nun die Module entsprechend ihrem Typ in den jeweiligen Speicherbereich.
5. Die alten Versionen des Loaders interpretieren die neuen Modul-Typen als sehr lange Files und verweigerten die Arbeit. (Fehler 103; nicht genug Arbeitsspeicher.)

12.2.4 Zukünftige Lösungen

Der Assembler und C-Compiler von Lattice werden voraussichtlich in naher Zukunft modifiziert werden, so daß diese Programme die neuen Modul-Typen bereits unter Programm-Kontrolle erstellen.

12.2.5 Das bewirken die neuen Bits

Die Modul-Größe ist ein Langwort, das die Anzahl der Langworte im Modul enthält. Für die in nächster Zukunft zu erwartenden Rechner mit 32-Bit-Adreßbus wurden die beiden höchstwertigen Bits freigehalten. Diese Bits wurden für ATOM folgendermaßen umdefiniert (Bit 31 heißt auch MEMF_FAST und Bit 32 MEMF_CHIP):

Bit 31	Bit 32	Bedeutung
0	0	Ist kein Bit gesetzt, wird der gerade zur Verfügung stehende Speicher belegt. Vorzug wird FAST-Memory gegeben. Diese Lösung ist aufwärtskompatibel.
1	0	Der LOADER muß FAST-Memory belegen oder einen Fehler ausgeben.
0	1	Der LOADER muß CHIP-Memory belegen oder einen Fehler ausgeben.
1	1	Sind beide Bits gesetzt, folgt diesem Langwort ein weiteres mit zusätzlichen Informationen. Diese Funktion ist noch nicht genutzt, sondern dient zukünftigen Erweiterungen.

12.2.6 Wirkung der Neuerung

Programme, die nicht mit den neuen Optionen assembliert oder kompiliert wurden, laufen (oder auch nicht) wie gewohnt. Das heißt aber auch, daß sie mit der Speichererweiterung in Konflikt kommen, wenn sie nicht sauber programmiert worden sind. Die alte Lösung steht weiterhin zur Verfügung.

Programmentwicklung kann (auch mit einer Maschine mit 512 Kbyte) wie gewohnt durchgeführt werden. Erst wenn das Programm nahezu fertig ist, sollte ATOM benutzt und dann das Programm auf einem Amiga mit mehr als 512 Kbyte RAM ausgetestet werden. Diese Lösung funktioniert auf allen drei Entwicklungs-Rechnern (Amiga, Sun und IBM).

Die mit ATOM entwickelten Programme sind unter V1.0 von Kickstart (auf dem Amiga 1000) nicht lauffähig. Hier wird der Fehler 103 (nicht genug Arbeitsspeicher) gemeldet.

OMD und DUMPOBJ in den Versionen V1.0 oder früher vertragen sich ebenfalls nicht mit ATOM-bearbeiteten Files.

12.2.7 Erforderliche Software

Zur Arbeit mit ATOM brauchen Sie folgende, zur Version 1.1 kompatible Software in der angegebenen oder einer späteren Version:

ATOM (Version 1.0)
Alink (Version 3.30)
Kickstart (Release 1.1) für den Amiga 1000.
DumpObj (Version 2.1) brauchen Sie nur, um mit ATOM bearbeitete Programme zu inspizieren.

12.2.8 Syntax der ATOM-Anweisung

Die Syntax für den Aufruf von ATOM ist:

```
ATOM <einfile> <ausfile> [-I]
```

oder

```
ATOM <einfile> <ausfile> [-C[C|D|B]] [-F[C|D|B]] [-P[C|D|B]]
```

Dabei stehen:

<einfile>	für das Objekt-File, das gerade assembliert oder kompiliert oder auch bereits mit ATOM behandelt wurde. (Ein File kann mehrmals »ATOMisiert« werden.)
<ausfile>	für das Ziel-File von ATOM
-I	für interaktiven Ablauf
-C	für *ändere Speicherbereich nach CHIP*
-F	für *ändere Speicherbereich nach FAST*
-P	für *ändere Speicherbereich nach PUBLIC* (der Typ Speicher, der gerade zur Verfügung steht)
-C	für *ändere nur CODE-Module*
-D	für *ändere nur DATA-Module*
-B	für *ändere nur BSS-Module*

12.2.9 Beispiele für ATOM-Aufrufe

Beispiel 1

In den meisten Fällen braucht der Code des Programmes nicht im Chip-RAM zu stehen. Der Blitter braucht nur den Inhalt von DATA- und BSS-Modulen. Die folgende Anweisung trägt dem Rechnung. Alle Daten und Definitionen werden in den Chip-RAM geladen, der eigentliche Code des Programmes kommt in Public-RAM. Geben Sie dazu ein:

```
ATOM einfile.obj ausfile.obj -pc -cdb
```

Beispiel 2

Um alle Module in den Chip-RAM zu laden, tippen Sie:

```
ATOM einfile.obj ausfile.obj -c
```

Beispiel 3

Um alle DATA-Module in den Chip-RAM zu laden, tippen Sie nun:

```
ATOM meinfile.o meinfile.at.o -cd
```

Beispiel 4

Hier ein Beispiel für den interaktiven Dialog mit ATOM. Die Eingabe des Anwenders wird in Kleinbuchstaben, die Ausgabe des Rechners in Großbuchstaben abgedruckt. In diesem Beispiel soll das CODE-Modul nach FAST, das DATA-Modul nach CHIP geladen werden. BSS-Module erscheinen nicht. Beachten Sie, daß zu Beginn HELP angefordert wird.

```
2> atom von.o von.set.o -i
AMIGA OBJECT MODIFIER V1.0
UNIT NAME FROM          ; Eingabefile
HUNK NAME NONE          ; kein Modulname
HUNK TYPE CODE          ; Modultyp ist CODE
MEMORY ALLOCATION PUPLIC ; Speicherbereich nicht extra angegeben
DISPLAY SYMBOLS [Y/N] y  ; Symbole aufzeigen ? ja
_basis
_xcovf.
_CXD22..
_druckf.
_haupt...
MEMORY TYPE? [F|C|P] ?   ; ? fordert Hilfe an
PLEASE ENTER F FOR FAST  ; Speicherarten
        C FOR CHIP
        P FOR PUPLIC
        Q TO QUIT        ; Programm beenden, kein Ausgabe-File wird erstellt
        W TO WINDUP      ; Verändert den Rest des Files nicht, übernimmt
        ihn in das Ausgabe-File
        N FOR NEXT HUNK  ; springt unmittelbar zum nächsten Modul
```

```
MEMORY TYPE? [F|C|P] f
UNIT NAME 0000
HUNKNAME NONE
HUNK TYPE DATA    ; Daten-Modul
MEMORY ALLOCATION PUPLIC

DISPLAY SYMBOLS? [Y/N] n
MEMORY TYPE? [F|C|P] c
UNIT NAME 0000
HUNKNAME NONE
HUNK TYPE BSS
MEMORY ALLOCATION PUPLIC
DISPLAY SYMBOLS? [Y/N] y  ; kein BSS-Modul !
MEMORY TYPE? [F|C|P] p
2>_
```

12.2.10 Fehlermeldungen

`Fehler Bad args (fehlerhafte Argumente)`

a) Eine Option beginnt nicht mit dem -
b) Falsche Anzahl an Parametern
c) Dem - folgt kein I, C, F oder P
d) -x und -I zusammen und so weiter

`Fehler Bad infile (unzulässiges Eingabefile)`

File wurde nicht gefunden.

`Fehler Bad outfile (unzulässiges Ausgabefile)`

Ausgabe-File kann nicht erstellt werden (Diskette voll, Schreibschutz aktiv und so weiter).

`Fehler Bad Type ## (Falscher Modul Typ)`

ATOM hat ein Modul mit nicht zulässigem Typ gelesen. Das Objekt-File kann defekt sein.

`Fehler empty input (Leeres Eingabefile)`

Das Eingabe-File enthält keine Daten.

`Fehler ReadExternals (Lese externe Symbole)`

Externe Definitionen oder Referenzen unbekannten Typs wurden gelesen. Das Eingabe-File kann zerstört sein.

`Fehler premature end of file (frühzeitiges End-of-file)`

Ein End-of-file (oder Out-of-data) wurde gefunden, obwohl weitere Daten erwartet wurden. Das Eingabe-File kann zerstört sein.

`Fehler This utility can only be used on files that have NOT been passed through ALINK (Dieses Programm kann nur noch nicht mit ALINK bearbeitete Files verarbeiten)`

Das Eingabe-File wurde bereits von ALINK bearbeitet, enthält also keine externen Referenzen mehr, alle Module wurden bereits verkettet. Mit ATOM können jedoch nur Ausgabe-Files eines Assemblers oder C-Compilers verarbeitet werden.

12.3 Ein neues Device erstellen und unter AmigaDOS einsetzen

Dieser Abschnitt vermittelt Ihnen die Informationen, die Sie zum Einbinden von Devices, die nicht Teil des AmigaDOS-Dateisystems sind, benötigen. Der nächste Abschnitt enthält dann Informationen zum Einbringen von Devices, die zum AmigaDOS-Dateisystem gehören sollen (Floppies, Festplatten und so weiter) – also alles Geräte, die Files und dazugehörige Directories schreiben und lesen.

Die Informationen dieses Abschnittes brauchen Sie, wenn zum Beispiel eine weitere serielle Schnittstelle in den Amiga eingebaut werden soll. In diesem Fall erstellen Sie ein Device mit dem Namen SER2:.

Dazu sind zwei Schritte notwendig. Zuerst erstellen Sie ein funktionsfähiges Hardware-Device, ein Vorgang, der hier nicht weiter erläutert werden kann.

Die Grundstruktur des Codes zum Erstellen eines Devices wird im Amiga ROM Kernal Manual gezeigt.

Dann wird das Device AmigaDOS unterstellt und verfügbar gemacht. Dieser Prozeß beinhaltet das Schreiben eines Device-Handlers (siehe *Amiga ROM Kernal Manual*) und das Installieren des Device in die Struktur von AmigaDOS.

Diese Installation geschieht durch die Erstellung einer geeigneten *Device-Node-Struktur* für Ihr Device. Diese sieht im wesentlichen wie eine DevInfo-Struktur für ein Diskettengerät aus. Lediglich das Startup-Argument kann von Ihnen selbst gewählt werden. Das SegList-Feld ist Null, der File-Name für Ihren Device-Handler kommt in das Feld Filename.

0	Nächstes
0	dt_device
0	Task (oder Prozeßkennung, siehe unten)
0	Lock
BSTR	File-Name oder Handler-Code
NNN	benötigte Stack-Größe
NN	benötigte Priorität
XXX	Startup-Information

0 SegList (ungleich Null, sobald der Code geladen wurde)
0 benötigter Globaler Vektor
BSTR Device-Name

Der Device-Handler ist das Interface zwischen Ihrem Device und dem Anwender-Programm. Er wird normalerweise in BCPL geschrieben. Der AmigaDOS-Kern versucht, diesen Code zu laden und einen neuen Prozeß für den Handler zu starten, wenn das Device zum ersten Mal angesprochen wird. Das geschieht automatisch, wenn der Kern bemerkt, daß das Task-Feld in der DevInfo-Struktur Null enthält. Ist der Code bereits geladen, steht der Wert des Code-Segment-Pointer im Feld SegList. Enthält dieses Feld Null, wird Code aus der Datei geladen, deren Name im Feld Filename steht, und das Feld SegList wird aktualisiert.

Dieser Prozeß des automatischen Ladens und Initialisierens wird von einem Code-Modul ausgeführt, das in BCPL oder in Assembler (im BCPL-Format) von Ihnen erstellt werden muß. Nur so kann das Kernal die Arbeit korrekt ausführen.

Schreiben Sie in Assembler, muß das Format des Codes dem unten gezeigten entsprechen. Sie brauchen DATA- und BSS-Sektionen, jedoch müssen beide in dem gezeigten Format geschrieben sein.

```
StartModule     DC.L    (EndModule-StartModule)/4
                        ; Größe des Moduls in Worten
Einsprungpunkt  .......
        .......  (Ihr Code)
        CNOP    0,4     ; Datengröße in Worten
        DC.L    0       ; Ende-Marke
        DC.L    1       ; Definiert Global 1
        DC.L    Einsprungpunkt-Startmodul      ; Offset zum Einsprungpunkt
        DC.L    1       ; Höchstes genutztes Global
        END
```

In Assembler schreiben Sie zu Beginn einen BCPL-Pointer zum initialen Packet, das vom Kernal vergeben wurde, in das Register D1.

Schreiben Sie in BCPL, finden Sie unten das Gerüst einer Routine. Die Hauptaufgabe des Device-Handlers ist das Umsetzen der Open-, Read-, Write- und Close-Zugriffe des DOS in das Format des Device. Alle anderen Packets werden mit einer Fehlermeldung zurückgewiesen.

```
GET "LIBHDR"
GET "IOHDR"
GET "MANHDR"
GET "EXECHDR"
```

Hier ein Grundgerüst für einen Handler-Task. Nach dem Erstellen des Task enthält das Parameter-Packet folgendes:

```
parm.pkt!pkt.arg1         =      BPTR zum BCPL-String des Devic-Namens, ("SKEL:")
parm.pkt!pkt.arg2         =      zusätzliche Informationen (wenn benötigt)
parm.pkt!pkt.arg3         =      BPTR zum Device Info Node
MANIFEST
$(
IO.blocksize = 30                ; Größe der IO Blocks des Devices
$)
LET start (parm.pkt) BE
$(LET extrainfo  =     parm.pkt!pkt.arg2
LET read.pkt     =     0
LET write.pkt    =     0
LET openstring   =     parm.pkt!pkt.arg1
LET inpkt        =     VEC pkt.res1
LET outpkt       =     VEC pkt.res1
LET IOB          =     VEC IO.blocksize
LET IOBO         =     VEC IO.blocksize
LET error        =     FALSE
LET devname      =     "serial.device*X00"
LET open         =     FALSE (Flag zum Anzeigen des Öffnungs-Status, mit
                       act.findinput oder act.findoutput)
LET node         =     parm.pkt!pkt.arg3
```

(Nun löschen wir erst den Block, um zu sehen, was später hineingeschrieben wurde, wenn OpenDevice aufgerufen wird.)

```
FOR i=0 TO Io.blocksize DO IOB!i := 0
IF OpenDevice ( IOB, devname, 0, 0) = 0 THEN error := TRUE
IF error THEN
$( returnpkt (parm.pkt;FLASE,error,objectinuse)
     return
$)
```

(Dann werden alle notwendigen Informationen in den Ausgabepuffer kopiert.)

```
FOR i=0 TO IO.blocksize DO IOBO!i := IOB!i
outpkt!pkt.type := act.write
inpkt!pkt.type := act.read
node!dev.task := taskid()              ; schreibt die Prozeßkennung in den Device-Node
```

(Das Parameter-Packet ist nun fertig und kann zurückgesandt werden.)

```
returnpkt (parm.pkt, TRUE)
```

(Die Hauptschleife wartet auf ein Ereignis.)

```
$( LET p = taskwait ()
SWITCHON p!pkt.type INTO
$(
CASE act.findinput:                ; OPEN
CASE act.findoutput:
$( LET scb = p!pkt.arg1
        open := TRUE
        scb!scb.id := TRUE                 ; INTERAKTIV
        returnpkt (p,TRUE)
        LOOP
$)
CASE act.end:              ; CLOSE
        node!dev.task := 0                 ; Prozeßkennung aus dem Device-Node löschen
        open := FALSE
        returnpkt (p,TRUE)
        LOOP
CASE act.read:                     ; Lesezugriff zurück
        inpkt := p
        handle.return (IOBO,read.pkt)
        LOOP
CASE act.write:            ; Schreibzugriff zurück
        outpkt := p
        handle.return (IOBO,write.pkt)
        LOOP
CASE 'R'                   ; Ein Lesezugriff
        read.pkt := p
        handle.request (IOB,IOC.read,p,inpkt)
        inpkt := 0
        LOOP
CASE 'W'                   ; Ein Schreibzugriff
        write.pkt := p
        handle.request (IOBO,IOC.read,p,outpkt)
        outpkt := 0
        LOOP
        DEFAULT:
        UNLESS open DO node|dev.task := 0  ; Prozeßkennung löschen wenn nicht geöffnet
        $)
        $) REPEATWHILE open | outpkt = 0 | inpkt = 0
        Termination
        CloseDevice (IOB)
```

```
LET buff = rp!pkt.arg2
LET len = rp!pkt.arg3
SetIO(IOB, command,?,rp!pkt.arg3,0)
putlong ( IOB, IO.data, buff )
SendIO(IOB,tp)

$(
LET errcode = IOB 0.error
LET len = getlong(IOB,IO.actual)
TEST errcode = 0 THEN                    ; kein Fehler
        returnpkt(p,len)
ELSE
        returnpkt(p,-1,errcode)
```

Schreiben Sie den Device-Handler in C, können Sie die Kernal-Routine des automatischen Ladens und der selbständigen Prozeß-Erstellung nicht verwenden. In diesem Fall laden Sie den Code selbst in den Rechner und rufen dann CreateProc auf, um den Prozeß zu erstellen. Das Ergebnis des Aufrufs wird dann im Task-Feld der DevInfo-Struktur abgelegt. Diese Mitteilung enthält außerdem noch weitere Daten, wie zum Beispiel die Nummer des Device. Der Handler-Prozeß sollte dann auf die einzelnen Zugriffe Open, Read, Write und Close warten, wie oben beschrieben. C-Code braucht auch nicht die Prozeßkennung in den Device Node zu schreiben, wenn er geladen wird.

12.4 Erstellen neuer Disk-Devices

Zum Erstellen eines neuen Disk-Device müssen Sie einen *Device-Node* konstruieren, wie es in Abschnitt 11.3.1 dieses Buches beschrieben wird. Dazu kommt dann noch ein Device-Treiber für das neue Disk-Device.

Ein Device-Treiber für ein neues Disk-Device muß die Aufrufe des Trackdisk-Device (beschrieben im *AmigaDOS ROM Kernal Manual*) nachahmen. Er muß in der Lage sein, alle Zugriffe wie Read, Write, Seek und Info genauso auszuführen, wie der Trackdisk-Treiber.

Die meisten Zeiger in der folgenden Beschreibung sind vom Typ BPTR (der in Kapitel 11 genauer beschrieben wird).

Konstruieren Sie den Node mit diesen Feldern:

0	Nächstes
0	dt_device
0	Task
0	Lock
0	Handler

210	Größe des Stack
10	Priorität
BPTR	auf Startup Info SegList
0	Globaler Vector
BSTR	auf Namen

Der BSTR auf den Namen ist ein BCPL-Pointer auf den Namen Ihres neuen Device (zum Beispiel HD0:). Dort finden Sie im ersten Byte die Länge des Namens, in den folgenden die Buchstaben des Namens selbst.

Die SegList muß die Segmentliste des Filing System Task sein. Um sie zu erstellen, müssen Sie ein Feld in der Process base in einem der Filing System Tasks dafür errichten.

Der folgende Code bewirkt das:

```
UBYTE *port;
port = DeviceProc( "DF0:");                ; Holt MsgPort vom Filing System Task
task = (struct Task * ) (port-sizeof(struct Task);; Taskstruktur ist unterhalb des Ports
list = ( task.pr_SegList )                 ; Erstellt Pointer vom SegArray
segl = list[3];          ; Drittes Element im SegArray ist die Filing-System-SegList
```

Dann müssen Sie die Startup Information setzen (denken Sie an die BPTR-Pointer). Diese Information besteht aus einem BPTR zu drei Langworten, die enthalten:

• die Gerätenummer (hier niemals Null verwenden !)

• den Device-Treiber-Namen, gespeichert als BPTR zum Device-Treiber-Namen, angeführt von einem Null-Byte, das mitgezählt wird (4/'H'/'D'/'O'/O).

Die *Disk-Size-Information* enhält folgende Felder vom Typ Langwort:

11	Größe der Tabelle
128	Größe eines Disketten-Blocks in Langworten (gleich 512 Byte)
0	Sektor Ursprung (erster Sektor ist Sektor Null)
Zahl der nutzbaren Flächen	2 bei einer Diskette
1	Anzahl der Sektoren pro Block
Anzahl Blocks je Spur	11 bei einer Diskette
2	(oder mehr) Anzahl der beim Start reservierten Blocks
0	Vorbelegte Blocks
0	Anzahl der freizuhaltenden Blocks

kleinste Zylindernummer	normalerweise 0
größte Zylindernummer	79 bei einer Diskette
5	(oder mehr) zeigt die Anzahl der Pufferblocks

Zuletzt muß dann der Device-Node an das Ende der Liste der Device-Nodes in der Info-Substruktur geschrieben werden. Beachten Sie, daß in den Next-Feldern BPTRs stehen.

Zum Partitionieren einer Festplatte brauchen Sie nur zwei oder mehr Device-Nodes mit (aufeinanderfolgenden) unterschiedlichen Einträgen für die erste und größte Zylindernummer einrichten.

12.5 AmigaDOS ohne Workbench/Intuition nutzen

Diese Informationen sollen dem Programmentwickler Informationen über die Interaktionen von AmigaDOS mit Intuition (beziehungsweise der Workbench) vermitteln. Intuition und die Input-Devices können jedoch nicht völlig umgangen werden. Sie können jedoch Ihre eigenen Input-Handler im Input-Stream installieren (wie im *Amiga-ROM-Kernel-Manual* unter *Input Devices* beschrieben) und Eingaben Ihren Wünschen entsprechend behandeln, nachdem Ihr Programm von AmigaDOS aus geladen und gestartet wurde. Schalten Sie Teile von Intuition aus, müssen Sie AmigaDOS zum Beispiel davor bewahren, dort nach den Routinen zum Verändern des Bildschirms zu suchen. Zudem muß Ihr Programm Wege zur Behandlung von Fehlern enthalten, da die Fehlerbehandlung mit Requestern ausfällt.

Ein weiterer Weg, den Amiga auf Ihre Weise zu nutzen, ist das Ignorieren des AmigaDOS-Dateisystems und der Einsatz des Trackdisk-Device, um Ihren Code und Ihre Daten in den Rechner zu laden. Details zu diesem Weg finden Sie im *Amiga-ROM-Kernel-Manual.*

Hier nun einige Informationen zu AmigaDOS und Intuition:

AmigaDOS initialisiert sich beim Laden selbst und öffnet dann Intuition. Dann versucht es, das mit Preferences erstellte Konfigurations-File (im DEVS-Directory) zu lesen und übergibt es an Intuition. Dann öffnet es über Intuition das erste CLI-Fenster und ist bereit, den ersten Befehl auszuführen. Normalerweise ist das LOADWB, gefolgt von ENDCLI.

Anstelle der Workbench kann aber auch ein Anwender-Programm stehen, für das ein neuer Prozeß erstellt wird. Der nächste Befehl ist dann ENDCLI, der alle Prozesse außer dem neuen Prozeß (und den Dateisystem-Prozessen) schließt. Dieser Prozeß würde das Feld pr_WindowPtr auf –1 setzen, was bedeutet, daß das AmigaDOS Fehler nicht mehr am Bildschirm meldet. Beachten Sie bitte, daß alle Fehler dann von Ihrem Programm bearbeitet werden müssen. Weitere Details hierzu finden Sie in Kapitel 11. AmigaDOS initialisiert auch das TrapHandler-Feld als Pointer auf den Code, der nach einem Fehler eine Meldung

ausgibt. Diese Routine muß durch eine von Ihnen geschriebene ersetzt werden. Damit sind alle Zugriffe des User-Task auf Intuition gesperrt. Wenn ein ernstes Problem mit dem Speicher auftritt, ruft AmigaDOS allerdings direkt Exec Alert auf.

Damit verbleibt noch das Problem, daß der Filesystem-Prozeß direkt Routinen aus Intuition aufruft, wenn ein Diskettenfehler oder ein Fehler im Filesystem auftritt, der durch einen Zusammenbruch des Arbeitsspeichers entsteht. Um das auch noch zu unterbinden, werden die Felder pr_WindowPtr- und tc_TrapHandler des Filesystem-Task auf –1 gesetzt und ein eigener TrapHandler bereitgestellt, wie unten beschrieben.

Den MsgPort eines Filesystems finden Sie, wenn Sie die Routine DeviceProc aufrufen und als Namen DF0, DF1 und so weiter einsetzen. Eine Fehlermeldung erscheint, wenn das Gerät nicht vorhanden ist. Vom MsgPort aus finden Sie die Basisadresse des jeweiligen Filesystem-Task. Damit haben Sie dann die beiden benötigten Felder und können den gewünschten Wert eintragen. So sollten Sie mit allen Disk-Einheiten verfahren.

Nun kann Ihr Programm Intuition schließen. (Die Workbench wurde erst gar nicht aufgerufen.) Dieser »Umweg« ist nötig, da es nicht möglich ist, das Öffnen von Intuition von AmigaDOS aus zu verhindern.

Soll Ihr Programm auf ein anderes Device Zugriff erlangen, so zum Beispiel auf SER:, muß dieser Handler ebenso verändert werden wie der Filesystem-Prozeß. Die Devices CON: und RAW: dürfen ohne Intuition natürlich nicht verwendet werden, da sie auf Intuition-Fenster zugreifen.

Stichwortverzeichnis

Bücher zum Amiga



M. Breuer
DELUXE
Grafik mit dem Amiga
1987, 370 Seiten
Eine behutsame Einführung in die grundlegenden Konzepte des jeweiligen Programms. Führt anhand kleiner überschaubarer Beispiele die wichtigsten Programmbefehle vor. Ein Nachschlageteil zu jedem Programm listet alle Befehle und ihre Bedeutung auf. Den Abschluß der Beschreibung jedes Programms bildet eine Sammlung von Tips und Tricks.
● Das deutsche Handbuch für den kreativen Grafikkünstler mit der DELUXE-Serie.
Best.-Nr. 90412
ISBN 3-89090-412-2
DM 49,–

Markt&Technik



A. Plenge
Amiga 3D-Grafik und Animation
1988, ca. 350 Seiten, inkl. Disk.
Angefangen bei den einfachsten Problemstellungen lernen Sie, professionelle 3D-Grafiken auf Ihrem Commodore Amiga zu planen, zu programmieren und darzustellen. Sämtliche theoretischen Grundlagen werden Ihnen anschaulich und leichtverständlich vermittelt. Auch scheinbar komplizierte Grafiken wie Schattenbildung, Reflexion, durchsichtige Gegenstände, Vielfachspiegelungen oder »raytracing« werden verständlich dargestellt.
● Eine ausführliche und leichtverständliche Anleitung für die Erstellung von dreidimensionalen Grafiken.
Best.-Nr. 90526
ISBN 3-89090-526-9
DM 69,–



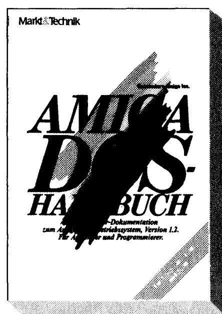
M. Kohlen
Grafik auf dem Amiga
1987, 337 Seiten
Dieses Buch enthält eine ausführliche Beschreibung der Grafik-Hard- und -Software, deren Funktionsweise und führt in die Grundzüge der Grafikprogrammierung ein. In den folgenden Kapiteln werden diese Kenntnisse dann in praktischen Beispielen umgesetzt. Außerdem bietet das Buch einen Überblick über die vorhandenen Soft- und Hardware-Erweiterungen für den Amiga.
● Eine Pflichtlektüre für jeden, der sich für die phantastische Grafik des Amiga interessiert.
Best.-Nr. 90236
ISBN 3-89090-236-7
DM 49,–



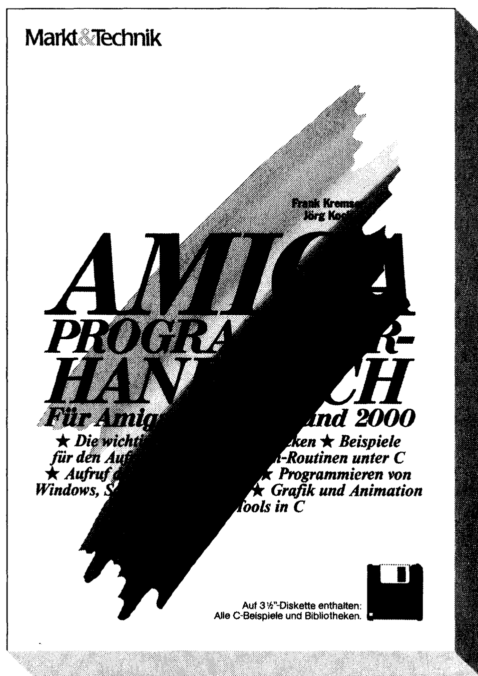
712300

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

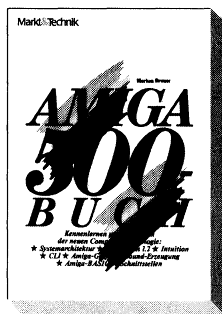
Bücher zum Amiga



Commodore-Amiga Inc.
Das Amiga-DOS-Handbuch für Amiga 500, 1000 und 2000
1988, 342 Seiten
Die Pflichtlektüre für jeden Commodore-Amiga-Anwender und Programmierer: eine Entwickler-Dokumentation zum Amiga-DOS-Betriebssystem, Version 1.2. Programmierung, interne Datenstruktur und Diskettenhandling. Mit diesem Buch lernen Sie das mächtige Amiga DOS schnell und sicher zu beherrschen. Alle Möglichkeiten des Systems, bis hin zum »Multi-Tasking« werden ausführlich und anschaulich beschrieben.
Best.-Nr. 90465
ISBN 3-89090-465-3
DM 59,-



Kremser/Koch
Amiga Programmierhandbuch
1987, 387 Seiten, inkl. Diskette
Eine tolle Einführung in die »Interna« des Amiga: Die wichtigsten Systembibliotheken, die das Betriebssystem zur Verfügung stellt, werden anhand vieler Beispiele erklärt. Aus dem Inhalt: Aufruf der Betriebssystem-Routinen unter C, Aufruf der DOS-Funktionen, Programmieren von Windows, Screens und Gadgets, Grafik und Animation, Tips und Tools in C.
Best.-Nr. 90491
ISBN 3-89090-491-2
DM 69,-



M. Breuer
Das Amiga-500-Handbuch
1987, 489 Seiten
Eine ausführliche Einführung in die Bedienung des Amiga 500. Kennenlernen und Anwenden der neuen Computer-Technologie: Systemarchitektur, Workbench 1.2, Intuition, CLI, Amiga-Grafik, Sound-Erzeugung, Amiga-BASIC und Schnittstellen. Neben dem Handbuchteil mit vielen Bildschirmfotos und Übersichtstabellen, die Ihnen beim täglichen Einsatz helfen, schnell und reibungslos zu arbeiten, enthält das Buch eine ausführliche Beschreibung des Amiga 500 und seines Zubehörs.
Best.-Nr. 90522
ISBN 3-89090-522-6
DM 49,-



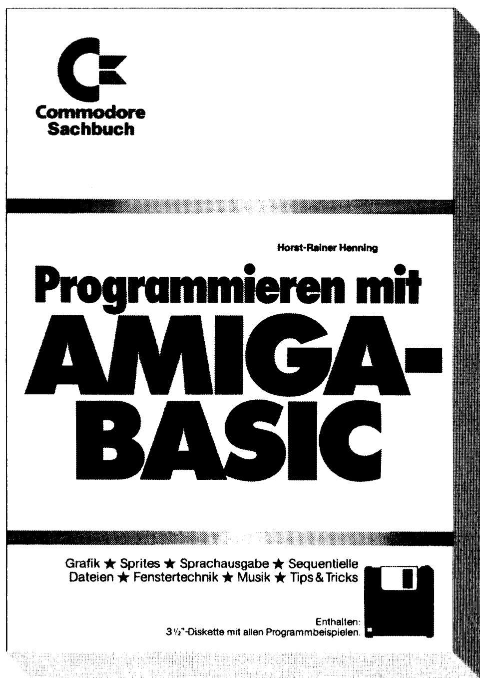
712328

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher zum Amiga

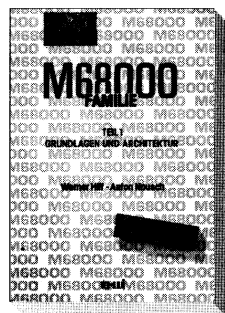


Prof. D. A. Lien
Amiga: Programmier-Praxis mit MS BASIC
1986, 394 Seiten
Bestseller! Eine systematische und lebendige Einführung in MS BASIC unter der komfortablen Mausbedienung und Fensteroberfläche des Amiga. Mit über 60 Musterprogrammen zu den Befehlen. Zeigt Amiga-typische Anwendungen: bewegte/farbige Grafiken; Musik- und Sprachausgabe, Strings, Felder, Mathematik, Dateibehandlung, Ein-/Ausgabe sowie »Entwurf von Programmen«.
Best.-Nr. 80369
ISBN 3-921803-69-1
DM 59,–



H.-R. Henning
Programmieren mit Amiga-BASIC
1987, 363 Seiten, inkl. Diskette
Einführung in die Programmierung des Amiga-BASIC: Grafik, Sprites, Sprachausgabe, sequentielle Dateien, Fenstertechnik, Musik, Tips und Tricks.

Hard- und Software-Anforderungen: Amiga 500, 1000 oder 2000 mit 512 Kbyte Arbeitsspeicher, gegebenenfalls ein grafikfähiger Matrixdrucker und ein Joystick, Amiga-BASIC von Microsoft
Best.-Nr. 90434,
ISBN 3-89090-434-3
DM 59,–



W. Hilf/A. Nausch
M68000-Familie: Teil 1 Grundlagen und Architektur
1984, 568 Seiten
Ausbildungs- und Entwicklungstext mit allen notwendigen Informationen über den M68000.
Best.-Nr. 80316
ISBN 3-921803-16-0
DM 79,–

W. Hilf/A. Nausch
M68000-Familie: Teil 2 Anwendungen und 68000-Bausteine
1985, 400 Seiten
In vielen Programmierbeispielen liefert dieses Buch die Praxis der in Teil 1 vermittelten Theorie.
Best.-Nr. 80330
ISBN 3-921803-30-6
DM 69,–



Zeitschriften · Bücher
Software · Schulung

712327

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0



in deutscher Sprache



Precision Software

Superbase

für den Amiga

mit mindestens 512 Kbyte RAM

Superbase – das relationale Datenbank-System

Superbase vereint als erstes Programm einer neuen Generation von Datenbank-Systemen sowohl eine neuartige, äußerst benutzerfreundliche Bedienung mit Pull-down-Menüs, Fenstern und Maussteuerung als auch die enorme Leistungsfähigkeit einer relationalen Dateiverwaltung.

Einfacher Datenbank-Aufbau

Mit den leichtverständlichen Pull-down-Menüs und Kontrollfeldern legen Sie in Minuten eine komplette Datenbank an. Sie können ein bereits festgelegtes Format jederzeit ändern, ohne Ihre Daten zu zerstören.

Verwaltung der Daten

Superbase zeigt Ihre Daten auf verschiedene Arten an, beispielsweise als Tabelle oder als Formular. Sind Index und Felder selektiert, so können Sie Ihre Daten wie bei einem Videorecorder anzeigen lassen. Schneller Vorlauf, Rücklauf, Pause und Stop – ein Recorder ist nicht einfacher zu bedienen. Ein einzigartiges Filtersystem wählt beliebige Datenkategorien aus, mit denen Sie dann arbeiten können.

Die Stärken von Superbase

Das Festlegen von Übersichten und zusammenhängenden Abfragen über mehrere verknüpfte Dateien ist auch bei verschiedenen Sortierkriterien kein Problem. Daten anderer Datenbanken oder Anwenderprogramme lassen sich ebenfalls problemlos verarbeiten. Binden Sie Daten in Ihre Textverarbeitung ein oder bilden Sie aus verschiedenen Dateien eine neue Datenbank! Die fortschrittliche Baumstruktur und die Disketten-Pufferung garantieren immer höchste Leistungsfähigkeit – Superbase findet beispielsweise einen bestimmten Datensatz in einer Datei, die 100 Adressen umfaßt, in nur 0,5 Sekunden.

Datenbank mit Bildern

Superbase bietet neben den gängigen Datenbank-Funktionen die Möglichkeit, Bilder und Grafiken darzustellen und zu verwalten. Einzigartigen Grafik-Datenbanken oder Dia-Shows steht also nichts im Wege.

Wer braucht Superbase?

Die Anwendungsmöglichkeiten sind nahezu unbegrenzt.

Hier einige Beispiele:

Geschäftliches	Professionelle Anwendungen
Lagerbestand	Design
Fakturierung	Fotografie
Registratur	Journalismus
Versandlisten	Sammlungen
Verwaltung	Forschung
Adressen	Ausbildung

* Unverbindliche Preisempfehlung

Leistungsumfang

Die Software: ● bis zu 17 Gigabyte Speicherkapazität pro Datei ● bis zu 16 Millionen Datensätze pro Datei ● maximal 999 Indizes pro Datei ● Anzahl der geöffneten Dateien, Anzahl der Dateien und Anzahl der Felder pro Datensatz: jeweils systemabhängig. Zum Beispiel: Für eine übliche Adreßverwaltung bei einer Datensatzlänge von 200 Byte können Sie auf Ihrer Diskette (880 Kbyte freier Speicher) ca. 4000 Adressen speichern.

Die Daten: ● Text, Daten, numerische Felder und externe Dateien ● Überprüfung bei der Eingabe ● Formelfelder ● Kalender der Jahre 1–9999, verschiedene Datumsformen ● verschiedene Zahlenformate bei 13stelliger Genauigkeit ● Datenschutz per Paßwort

Die Ausgaben: ● das Programm beherrscht einen flexiblen Etikettendruck und produziert übersichtliche Listen mit dem Reportgenerator ● bis zu 255 Spalten ● mit Titel, Datum und Seitenzahl ● Datensatz-Zähler, Durchschnitt, Zwischen- und Endergebnis ● Ausgabe von mehreren Dateien auf Bildschirm, Drucker, Diskette oder neuer Datei ● mehrspaltiger Etikettendruck mit variablem Format ● Speicherung der Ausgabe- und Abfrage-Formate zur späteren Verwendung ● vielfältige Sortierkriterien

Hardware-Anforderung

Amiga mit mindestens 512 Kbyte RAM, beliebiger Drucker mit Centronics-Schnittstelle.

Best.-Nr. 51636

DM 249,–* (sFr 199,–*/öS 2490,–*)

Übrigens: Superbase gibt es auch für Atari ST, Schneider PC und IBM-PCs und Kompatible



70226

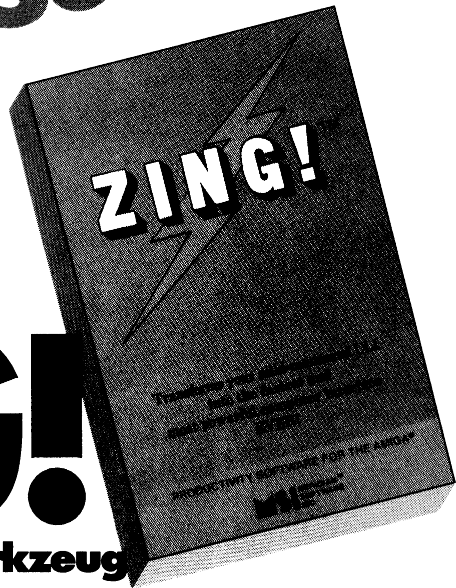
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bestellungen im Ausland bitte an: SCHWEIZ: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Telefon (042) 415656 · ÖSTERREICH: Rudolf Lechner & Sohn, Heizwerkstraße 10, A-1232 Wien, Telefon (0222) 677526 · Ueberreuter Media Verlagsges. mbH (Großhandel), Laudongasse 29, A-1082 Wien, Telefon (0222) 481543-0.

Brandneue

stop · unentbehrlich für jeden Amiga-User · stop · frisch bei Markt&Technik eingetroffen · stop · deutsche Programmversionen in Arbeit · stop · exklusiv bei Markt&Technik · stop · Update-Service für alle unsere Kunden · stop

Amiga-Software



ZING!

Das mächtige CLI-Werkzeug

Mit ZING! haben Sie endlich das gesamte File-System mit Directories und Subdirectories fest im Griff. Sie beschleunigen mit ZING! alle nötigen Arbeiten mit Files, verwalten bis zu 500 Files und Subfiles und bis zu 100 Directories auf einmal.
Die Bedieneroberfläche ist vom Feinsten:
- Pull-down-Menüs
- (Click-)Icons
- Funktionstasten

Weitere Optionen wie: Task-Monitor, Printer-Spooler, Screen-Syer/Printer, Screen-Dimmer, Veränderung der Voreinstellung der Funktionstasten und des Systems.
Am besten gleich bestellen!

Best.-Nr. 52571

DM 189,–*

*Unverbindliche Preisempfehlung.

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

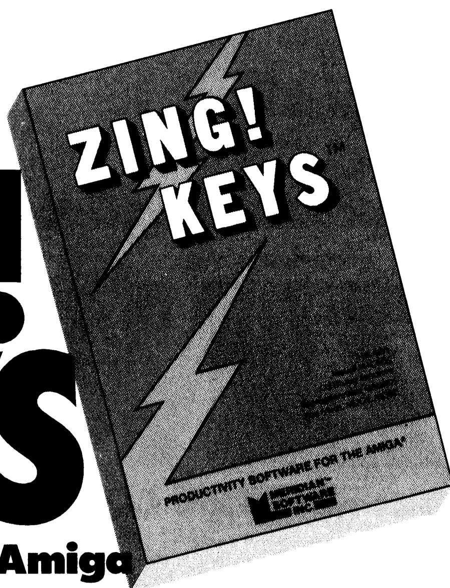
707324

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Brandneue Amiga-Software

stop · unentbehrlich für jeden Amiga-User · stop · frisch bei Markt&Technik eingetroffen · stop · deutsche Programmversionen in Arbeit · stop · exklusiv bei Markt&Technik · stop · Update-Service für alle unsere Kunden · stop

ZING! KEYS



Ihr ganz persönlicher Amiga

Mit ZING!KEYS machen Sie aus Ihrem Amiga das variable System, das Sie sich schon immer wünschen. Es ist Ihren eigenen Ansprüchen jederzeit anpaßbar! Alle Tasten sind nach Wunsch belegbar: z.B. mit Funktionsaufrufen, Programmaufrufen, Systembefehlen und vorprogrammierten Befehlen. Die Belegung ist natürlich jederzeit abspeicherbar.

Durch die Belegung von »Hot-Keys« haben Sie mit ZING! KEYS ein Multitaskingsystem par excellence!

Best.-Nr. 52572

DM 109,–*

*Unverbindliche Preisempfehlung.

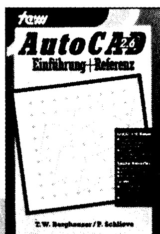
Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

707325

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

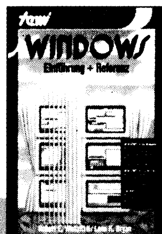
PC-Spezial-Literatur von tewi



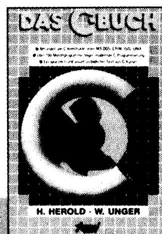
DM 79,–



DM 79,–



DM 79,–



DM 79,–



DM 49,–



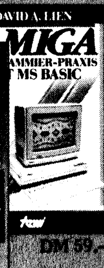
DM 59,–



DM 59,–



DM 69,–



DM 59,–



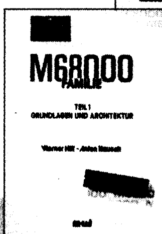
DM 39,–



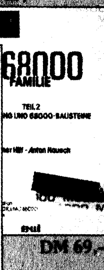
DM 59,–



DM 39,–



DM 79,–



DM 69,–



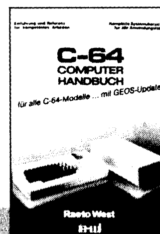
DM 79,–



DM 59,–
DM 79,–



DM 49,–



DM 66,–

tm 5090

Fordern Sie unser neues Gesamtverzeichnis an!

tewi Verlag GmbH
Theo-Prosel-Weg 1
8000 München 40

Bitte schneiden Sie diesen Coupon aus, und schicken Sie ihn in einem Kuvert an:
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar

Computerliteratur und Software vom Spezialisten

Vom Einsteigerbuch für den Heim- oder Personalcomputer-Neuling über professionelle Programmierhandbücher bis hin zum Elektronikbuch bieten wir Ihnen interessante und topaktuelle Titel für

• Apple-Computer • Atari-Computer • Commodore 64/128/16/116/Plus 4 • Schneider-Computer • IBM-PC, XT und Kompatible
sowie zu den Fachbereichen Programmiersprachen • Betriebssysteme (CP/M, MS-DOS, Unix, Z80) • Textverarbeitung • Datenbanksysteme • Tabellenkalkulation • Integrierte Software • Mikroprozessoren • Schulungen.
Außerdem finden Sie professionelle Spitzen-Programme in unserem preiswerten Software-Angebot für Amiga, Atari ST, Commodore 128, 128 D, 64, 16, für Schneider-Computer und für IBM-PCs und Kompatible!
Fordern Sie mit dem nebenstehenden Coupon unser neuestes Gesamtverzeichnis und unsere Programmservice-Übersichten an, mit hilfreichen Utilities, professionellen Anwendungen oder packenden Computerspielen!



Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2,
8013 Haar bei München, Telefon (089) 4613-0

Bitte schicken Sie mir:
☐ Ihr neuestes Gesamtverzeichnis
☐ Eine Übersicht Ihres Programmservice-Angebotes aus der Zeitschrift ____________
☐ Außerdem interessiere ich mich für folgende/n Computer: ____________

(PS: Wir speichern Ihre Daten und verpflichten uns zur Einhaltung des Bundesdatenschutzgesetzes)

Adresse:

Name ____________

____

Straße ____________

Ort ____________

Markt&Technik Verlag AG
– Unternehmensbereich Buchverlag –
Hans-Pinsel-Straße 2
D-8013 Haar bei München

Amiga DOS-Handbuch

Wer alles über das moderne Multi-Tasking-Betriebssystem des Amigas wissen möchte, für den wird das AmigaDOS-Handbuch zu einer unentbehrlichen Pflichtlektüre werden! Das Autorenteam – es handelt sich dabei um die Entwickler des Amiga-Betriebssystems – haben Ihr Wissen in dieser einzigartigen Dokumentation dargelegt. Sie enthält drei – im amerikanischen Original ursprünglich getrennt veröffentlichte – Bücher:

- Das AmigaDOS-Anwender-Handbuch
- Das AmigaDOS-Programmierer-Handbuch und
- Das technische AmigaDOS-Handbuch

Im **Anwender-Handbuch** finden Sie Informationen, die für alle Benutzer – also auch Einsteiger – von elementarer Wichtigkeit sind. Der Amiga versteht nämlich weit mehr Instruktionen, als über die »Workbench« per Maus erreichbar sind. Diese Instruktionen können Sie verwenden, wenn Sie das sogenannte »CLI« (Command Line Interface) aktivieren. Alle Befehle, die AmigaDOS Ihnen in dieser Ebene nun zur Verfügung stellt sind ausführlich erklärt und mit Beispielen unterlegt. Besonders wird auf die Anwendung der Editoren »ED« und »EDIT« eingegangen, die Sie für das Schreiben von individuellen Batch-Programmen benötigen.

Das **Programmierer-Handbuch** dagegen beschreibt, wie die AmigaDOS-Funktionen in selbst erstellten Programmen genutzt werden können. Daneben wird die Assembler-Programmierung mit dem Amiga-Macro-Assembler und des dazugehörenden Linkers erklärt. Für das Verständnis dieses Teils sind gute 68000er-Kenntnisse erforderlich.

Das **Technische AmigaDOS-Handbuch** vermittelt Ihnen den Aufbau und die Anwendung der im Betriebssystem intern verwendeten Datenstrukturen des AmigaDOS. Eine Beschreibung des Disketten-Aufzeichnungsformates fehlt ebensowenig wie Erklärungen zum Format der sogenannten »Objekt-Files« des AmigaDOS. Software-Entwickler oder interessierte Anwender werden in diesem Buch Antworten auf viele programmtechnische Fragen finden.

Hardware-Anforderungen:
Amiga 500, 1000 oder 2000

Software-Anforderungen:
Für das Nachvollziehen der Beispiele im Programmierer-Handbuch ist ein Assemblersystem für den Amiga erforderlich (KSeka, DevPac oder Metacomco Makro-Assembler).

ISB N 3-89090-465-3





DM 59,-
sFr 54,30
öS 460,20